UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

Implementação e Análise de Desempenho do *Framework Apache Hadoop* e da Biblioteca *OpenMPI* para Multiplicação de Matrizes com um *Cluster* de Baixo Custo na Plataforma *Raspberry PI*

Trabalho de Conclusão de Curso

Elias Rabelo Matos

São Cristóvão – Sergipe

2018

UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

Elias Rabelo Matos

Implementação e Análise de Desempenho do *Framework Apache Hadoop* e da Biblioteca *OpenMPI* para Multiplicação de Matrizes com um *Cluster* de Baixo Custo na Plataforma *Raspberry PI*

Trabalho de Conclusão de Curso submetido ao Departamento de Computação da Universidade Federal de Sergipe como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador(a): Kalil Araujo Bispo
Coorientador(a): Edward David Moreno Ordonez

São Cristóvão – Sergipe

2018

Para José Martins Matos

Agradecimentos

Quero agradecer primeiramente à minha família, à minha mãe, Lindicelma, mulher guerreira e educadora por ter me ensinado a importância dos estudos. Ao meu pai, José Araujo, por ter me ensinado tudo que não estava ligado à colégio ou faculdade. Quero agradecer a minha irmã, Ana Maria (aia), por todas as vezes que brigamos e sorrimos juntos durante toda a vida. Quero agradecer a minha avó Josefa por ser minha mãe do mesmo jeito que minha mãe. Agradeço aquele a quem dedico este trabalho e quem eu queria que estivesse aqui, meu avô José Martins, nunca esqueço das suas histórias, nem dos seus conselhos, nem paro de imaginar seu orgulho em me ver formado mesmo que não entendesse muito bem que curso era o meu.

Após a minha família gostaria de agradecer a todos aqueles que compartilharam seus ensinamentos comigo, desde colégio até agora, mesmo aquelas aulas que eu odiei. Gostaria de agradecer a todos os autores de artigos, livros, ou qualquer outra fonte que me ajudaram a produzir este trabalho. Gostaria de agradecer a professora Beatriz Trinchão por ter sido minha primeira professora de programação e me ajudado a construir todo o alicerce do curso. Gostaria de agradecer ao professor Bruno Prado por ter, mesmo com seus exemplos sem muito sentido, ter me ensinado tanto em dois anos e meio de PRODAP. Ao meu orientador Kalil, gostaria de agradecer por ter aceitado me orientar mesmo estando lá em Portugal preocupado com seu Pós-Doutorado, espero ter sido digno de sua orientação. Kalil foi meu professor apenas em Fundamentos da Computação no já longínquo primeiro período, ainda sim, um dos que mais me ensinaram com todas suas aparições nas reuniões do PRODAP. Meu coorientador Edward, obrigado por ter me ajudado de perto na ausência física de Kalil, obrigado por ser uma inspiração na área de Computação de Alto Desempenho para qualquer um e por ser um dos professores mais divertidos que já tive.

Obrigado também ao professor Marco Túlio por ter me apresentado a plataforma *Raspberry PI* e a Felipe dos Anjos por ter sido sempre solicito para tirar dúvidas sobre o *OpenMPI*. E obrigado, também a Marco e Felipe por comporem a banca desse TCC.

Por ultimo gostaria de agradecer a meus velhos amigos, Guiga, Hiago, Luis, Daniel, Leo, Dalton, Neto e Vanessa. Obrigados a todos meus amigos de Antas, Cícero e Fátima por terem compartilhado os melhores momentos da minha vida. Obrigado especialmente à Antônio Carlos por ter sido um elo entre meus amigos de antas e meus amigos da faculdade. Entre meus amigos da faculdade obrigado especialmente à Patrick Jones, José Joaquim, Antônio Bispo, João Matheus, Adrian Costa, Jackson Tavares, Artur Santos, Felipe Pereira, Hugo Barreto, Natanael Batista, Alana Lúcia, Jusley Arley, Givaldo Marques, Itamar Sousa, Werthen Castro, Italo Jorge, Luiz Gabriel, Thiago Fontes, Edgar Vieira, João Manoel, Gabriel Leite, Diego Andrade e todos do amigos que fiz nesses quatro anos e meio, inclusive de outros cursos.

And when it's over I'd just as soon go on my way
Up to some paradise
Where the trout streams flow and the air is nice
And ride a horse along a trail
(Hurricane - Bob Dylan)

Se o senhor nunca viveu num ambiente
onde novas coisas são criadas o tempo
inteiro, não vai entender nunca o quanto
isso é formidável.
(Estilo Marciano - Isaac Assimov)

Nove vezes sete, pensou Shuman com orgulho,
sessenta e três. Não precisava mais que um
computador lhe dissesse isso. Sua própria
cabeça era um computador. E isso lhe dava
uma fantástica sensação de poder.
(A Sensação de Poder - Isaac Assimov)

Resumo

A Computação de Alto Desempenho (HPC) é responsável por pesquisar o processamento de dados em um tempo menor possível. A estrutura da HPC sempre foi muito complexa, muitas da vezes custando milhares de dólares para a construção de um *cluster*. Devido ao alto custo, a HPC esteve por muito tempo concentrada em universidades, instituições militares ou grandes empresas. No entanto, nos últimos anos, com a popularização de Sistemas Embarcados com capacidade de processamento significativa e seguindo a Arquitetura ARM, observou-se uma possibilidade de uso desses sistemas para HPC com desempenho significativo e um baixo custo. Portanto, este trabalho de conclusão de curso se faz análise de desempenho de algoritmos de multiplicação de matrizes executando em um *cluster* embarcado de baixo custo na plataforma *Raspberry PI*, usando o *framework Apache Hadoop* e a biblioteca *OpenMPI*. Detectou-se um baixo desempenho do *Apache Hadoop* em relação ao *OpenMPI* em algoritmos de multiplicação de matrizes. E que a linguagem Java é dezenas de vezes mais lenta do que a linguagem C.

Palavras-chave: Computação de Alto Desempenho, *Cluster*,Sistema Distribuído, Avaliação de Desempenho, Arquitetura ARM.

Abstract

The High Performance Computing (HPC) is the research area responsable for processing data in less time possible. HPC always have a very complex structure, were the cost for build a cluster is over thousands dollars. Because of the high cost, the HPC was been for a long time restricted to universities, military institutions and large companies. However, in recent years, Embedded Systems using ARM Architecture with a satisfactory computing power and low cost, became popular. Then was identified a way possible to make HPC with a low cost and significative power. On this way this work we make a design analysis of matrix multiplication schemes in a low cost embedded cluster in the Raspberry PI platform, using the Apache Hadoop framework and an OpenMPI library.The low performance of Apache Hadoop over OpenMPI was detected in matrix multiplication algorithms. And that is a Java language is dozens times slower than C language.

Keywords:High Performance Computing,cluster, Embebbed System,ARM Architecture,Performance Evaluation

Lista de ilustrações

Lista de tabelas

Lista de Pseudocódigos

Lista de códigos

Lista de abreviaturas e siglas

HPC	High Performance Computing
HDFS	Hadoop Distributed File System
MPI	Message Passing Interface
RISC	Reduced Instruction Set Computer
CISC	Complex Instruction Set Computer
ARM	Advanced RISC Machine
MPIS	Millions Instructions Per Second
TCP/IP	Transmission Control Protocol/Internet Protocol
FLOPS	FLoating-point Operations
HPL	High-Performance Linpack benchmark
HIPI	Hadoop Image Processing Interface

Sumário

1

Introdução

A *Máquina de Turing*, proposta inicialmente em 1936, foi o primeiro modelo teórico para computador programável criado (SIPSER, 1996).(No contexto da segunda Grande Guerra, o próprio Alan Turing trabalhou para os aliados). Turing liderava uma equipe cujo objetivo era quebrar os códigos enviados pela Alemanha Nazista pela máquina chamada de Enigma. Em termos gerais, Turing e sua equipe desenvolveram uma máquina chamada de *The Bomb* (SALE, 2010), que processava os dados capturados das transmissões nazistas e tentava decodificá-los. No entanto a quantidade de combinações necessárias para obter as mensagens cifradas era maior do que a quantidade de tentativas para encontrar a chave necessária para ler a mensagem que a *The Bomb* podia realizar em um dia. Desta maneira quando o processamento estivesse terminado, as ordens contidas na mensagem já teriam sido executadas.

A partir da máquina eletrônica desenvolvida por Turing, o que, em termos históricos, representa o início da Ciência da Computação, a necessidade de um grande poder de processamento sempre foi inerente a computação, com a necessidade de processamento cada vez maior. Um problema que persistiria até os dias atuais surgia: Dado uma quantidade de dados, como processá-los em um tempo que tal resposta seja útil? Desse modo os dados que precisavam ser calculados nos computadores primordiais eram muito menores do que os dados calculados atualmente em computadores atuais. No entanto, o poder computacional disponível naquela época era muito pequeno. Para se ter uma ideia uma comparação feita pelo blog (ANTIQUETECH, 2013), entre o ENIAC, comumente reconhecido como o primeiro computador da história e um telefone celular Samsumg SCH-A850, constatou-se que a placa mãe do celular custa dezessete mil vezes menos, usa quatrocentas mil vezes menos energia e é mil e trezentas vezes mais rápido.

Pesquisas desenvolvidas na área de desempenho computacional mostram o quanto a velocidade das placas mãe foi multiplicada.No entanto a quantidade disponível de dados também cresceu exponencialmente. Ao referi-se à quantidade de dados disponíveis não tange apenas a dados que precisam ser armazenados. Mas, sobretudo, a dados que precisam ser processados.

Tais dados não se referem apenas a propósitos militares e/ou acadêmicos como no início dos primeiros computadores. Os dados que precisam ser calculados e analisados referem-se a relações interpessoais e econômicas da sociedade cotidiana. Os exemplos são muitos: dados são processados para garantir uma melhor logística em empresas de transportes, dados são processados para um fluxo melhor da rede de computadores utilizada diariamente pelas pessoas, a partir dos dados coletados dos usuários é possível calcular e processar possíveis produtos de interesse e gerar uma propaganda, entre outros.

Com a crescente disponibilidade de dados, e a inerente necessidade de analisar e classificar os mesmos, muitas das vezes dependendo da realização de cálculos cada vez mais complexos, isso dá origem a uma área da computação chamada de Computação de Alto Desempenho (do inglês: *High Performace Computing*)(HOFFMAN et al., 1990). Para HOFFMAN et al. (1990), essa área deve pesquisar de quais maneiras um determinado problema deve ser abordado para que se chegue o mais rápido possível a um resultado. É importante salientar que os referidos problemas baseiam-se principalmente na resolução de cálculos.

A forma primordial de execução de um programa de computador é a forma sequencial, no entanto, muitos dos programas que são desenvolvidos podem ser executados de forma paralela, isto é, o programa pode ser dividido de modo que subpartes do programa sejam executadas de forma simultânea. No entanto, um computador comum não dispõe de um paralelismo de fato, onde mais de uma CPU torna possível realizar duas ou mais tarefas simultaneamente e sim de um pseudoparalelismo, onde a CPU faz um chaveamente entre tarefas (TANENBAUM; BOS, 2014). Fazer o uso do paralelismo é chave primordial para o ganho de tempo na Computação de Alto Desempenho.

Para eliminar o problema causado pela incapacidade de um paralelismo real na Computação de Alto Desempenho, são comumente utilizados os chamados *clusters* ou aglomerado de computadores. A ideia é antiga e remonta a década de 60, mas inicialmente o custo de construção era muito alto e os *clusters* tiveram por muito tempo em posse de academias, empresas ou instituições militares (PFISTER, 1998).

1.1 Motivação

Os primeiros *mainframes* já foram interligados, ainda na década de 60, formando os primeiros *clusters*. A técnica de *clustering* ganhou uma relevância maior na área da computação na década de 80, devido a crescente popularização de microcomputadores. No entanto, durante muito tempo essa técnica esteve ligada a organizações militares, universidades e grandes empresas devido ao alto custo de produção(PFISTER, 1998).

Com a ideia que a Computação de Alto Desempenho possa ser uma área da computação com uma presença maior na atividade cotidiana de estudantes de computação e de profissionais da área, este trabalho busca uma forma acessível de pesquisar nessa área mesmo com recursos

escassos. Como o custo é um fator decisivo, novas formas de construir um *cluster* com um custo mais acessível serão buscadas. Em 2012, a *Raspberry PI fundation* lançou o primeiro modelo do Raspberry PI sob o slogan "Um computador do tamanho de um cartão de crédito"[1].

Ao custo de 25 dólares em 2013, o modelo inicial do Raspberry PI foi o Model A, contando com um hardware formado por um processador ARMv6 (32 *bits*, *single core*), 256MB de memoria RAM, placa *ethernet* de 100MB, um cartão SD funcionando como memória de armazenamento e uma série de dispositivos de entrada e saída. O lançamento do Model A foi seguido por subsequentes modelos contando com alguns incrementos no hardware. O modelo mais potente disponível atualmente é o Model B 3, que conta com um ARMv8v-A (64 *bits, quad-core*) e 1GB de memória RAM, que é vendido por 35 dólares em 2018[2].

O objetivo principal da *Raspberry PI Fundation* com o lançamento destes computadores de baixo custo era inicialmente estimular o ensino de programação em escolas. Mas mesmo sendo frequentemente confundido com um brinquedo, os *Raspberry* apresentam um poder computacional que permite realizar as atividades de um usuário comum. Mesmo contando com limitações como a falta de *softwares* disponíveis para a arquitetura dos processadores ARM existe uma gama crescente de possibilidades de uso desses computadores de forma profissional e/ou acadêmica. Uma dessas possibilidades é a construção de um *cluster* para computação de alto desempenho.

1.2 Objetivos

O presente trabalho tem como objetivo principal a construção de um *cluster* baseado em *Raspberry PI* de forma que mostre-se a possibilidade de utilização de tais computadores de baixo custo na realização de Computação de Alto Desempenho.

Como objetivos especificos tem-se: A realização de uma revisão sistemática acerca do estado da arte no assunto de Computação de Alto Desempenho em *Raspberry Pi* e a construção do *cluster* de Raspberry PI de duas maneiras distintas e simultâneas, uma usando a plataforma *Hadoop* e outra usando a biblioteca *OpenMPI*. Ambas as formas têm ênfase no paralelismo, de modo que nesse trabalho será executado o mesmo problema em cada uma das opções para então analisar o desempenho de ambas alternativas e por fim também fazer uma analise comparativa.

Também é importante mostrar a gama de opções disponíveis com uso de *Raspberry PI* na construção de *clusters* em geral e não só o uso de *cluster* para Computação de Alto Desempenho. Essas opções serão vistas no restante do documento.

[1] Tradução live de "A credit card size computer", anteriormente usado no site oficial da empresa: raspberrypi.org
[2] Especificações completas disponíveis em: raspberrypi.org/products

1.3 Metodologia

O presente trabalho norteou-se pela realização de uma revisão sistemática tal como é definida por Kitchenham (2004), de forma que inicialmente serão analisados todos os trabalhos relacionados a construção de *cluster Raspberry PI*.

No capítulo 2 foi feita uma revisão teórica acerca dos temas necessário ao longo do trabalho. Em seguida no capítulo 3, foi realizada a revisão sistemática. A partir da revisão realizada foi possível extrair dos trabalhos relacionados, um objeto de estudo para o desenvolvimento da parte prática do trabalho.

1.4 Organização do Documento

O documento está organizado da seguinte forma:

- Capítulo 1- Introdução: Uma visão geral sobre a área e descrição do objetivo do trabalho;
- Capítulo 2 - Revisão Bibliográfica: Principais conceitos para a compreensão de algumas partes desta pesquisa;
- Capitulo 3 - Trabalhos Relacionados: Uma revisão sistemática;
- Capítulo 4 - Multiplicação de Matrizes: Explicação sobre os experimentos realizados;
- Capítulo 5 - Implementação de MM no *Hadoop* e *OpenMPI*: A forma que os experimentos foram implementados;
- Capitulo 6 - Resultados: O que foi obtido a partir dos experimentos;
- Capítulo 7- Conclusão: Considerações Finais.

2

Revisão Bibliográfica

Neste capitulo será apresentada uma revisão bibliográfica acerca de assuntos fundamentais para o desenvolvimento do restante do trabalho. A intenção deste capítulo é fazer uma revisão acerca dos tópicos que foram necessários para a construção do presente trabalho. É importante destacar que o presente capítulo não abordará profundamente nenhum dos tópicos e sim trará a compreensão necessária o decorrer do texto.

O capítulo será dividido em sete seções, sendo elas: A Seção 2.1 traz o conceito de paralelismo necessário no decorrer do documento. Seguido em 2.2 por considerações sobre arquitetura de computadores. Prosseguindo a Seção 2.3 introduz o conceito de sistema distribuído. Com essas definições, pode-se na Seção 2.4 introduzir o conceito de *cluster*. Por fim, a Seção 2.5 introduz o conceito de Computação de Alto Desempenho. Finalizada a revisão sobre conceitos clássicos da computação que foram abordados neste trabalho, o capítulo se encerra nas Seções 2.6 e 2.7 introduzindo respectivamente o *Apache Hadoop* e *OpenMPI* que serão as ferramentas utilizadas no desenvolvimento dos experimentos.

2.1 Paralelismo

Paralelismo significa na língua portuguesa "Correspondência entre duas coisas ou situações."[1] Para a ciência da computação o paralelismo é utilizado para definir a execução de mais de uma tarefa ao mesmo tempo pelo computador, sendo estas tarefas iguais ou não(TANENBAUM; BOS, 2014).

Para Tanenbaum e Bos (2014), a fase da multiprogramação é onde a CPU alterna entre vários programas a cada alguns milissegundos gerando o pseudoparalelismo. Para Tanenbaum e Bos (2014) a multiprogramação é a terceira geração dos sistemas operacionais, nessa fase foi detectado que muito do tempo de execução do programa era gasto em operações de entrada e de

[1] Disponível em: https://www.dicio.com.br/paralelismo/

saída e, enquanto essas operações eram executadas, a CPU ficava ociosa. Percebido esse tempo ocioso, os sistemas operacionais começaram a fazer um escalonamento de forma que enquanto um processo estivesse executando uma operação de entrada e de saída, outro pudesse usar a CPU para realizar processamento.

A multiprogramação representou um ganho enorme de desempenho nos computadores das décadas de 60 e 70, com essa possibilidade de intercalar processos um computador foi capaz de realizar mais de uma tarefa por vez. Essa característica impulsionou a popularização dos computadores nas décadas subsequentes. A troca de processos realizada pela CPU tornou-se cada vez mais sofisticada de modo que dezenas, centenas de processos pudessem ser intercalados. Tal característica tornou comum a crença de que o computador executa diversas tarefas por vez, mas é preciso ao falar de execução simultânea em computação distinguir corretamente o que é o paralelismo de fato do pseudoparalelismo.

2.1.1 Pseudoparalelismo

> *In any multiprogramming system, the CPU switches from process to process quickly,running each for tens or hundreds of milliseconds. While, strictly speaking, at anyone instant the CPU is running only one process, in the course of 1 second it may work on several of them, giving the illusion of parallelism. Sometimes people speak of pseudoparallelism in this context, to contrast it with the true hard-ware parallelism of multiprocessor systems (which have two or more CPUs sharing the same physical memory) (TANENBAUM; BOS, 2014, pp. 86).*

Ao falar da capacidade de um computador executar diversos programas de maneira simultânea o que provavelmente esteja sendo falado não seja de a execução de mais de uma tarefa pela CPU, possibilidade disponível em sistemas multiprocessados como exposto por Tanenbaum e Bos (2014) no epigrafe desta seção. O pseudo paralelismo não é uma característica de *hardware* e sim de Sistema Operacional.

Pela forma que um sistema operacional executa um programa, ele cria uma hierarquia de processos, ou seja, cada processo tem uma prioridade de execução definida pelo sistema operacional. A forma que o sistema operacional intercala seus processos varia entre as implementações, mas uma coisa permanece presente. O tempo que cada processo tem de CPU é de um tempo muito menor do que o perceptível por um ser humano. Então cada um dos processos que estão na fila de execução têm o seu tempo de uso de uma forma que transparece a ideia que todos eles estão sendo executados simultaneamente o que se confunde com paralelismo.

2.1.2 Paralelismo

É importante observar que no contexto deste trabalho o pseudoparalelismo não é objeto de estudo e todas as referências ao termo *paralelismo* referem-se a paralelismo como apresentado nesta seção ou sistemas multiprogramados. No passado quando se falava em melhoria de

performance era comum o aumento da capacidade de *clock* do processador. No entanto por limitações físicas esta solução não é mais suficiente.

A solução para isso foi o paralelismo, ao contrário do pseudoparalelismo, onde os processos são intercalados na CPU. O paralelismo acontece em sistemas que contam com mais de uma CPU, as CPUs de sistemas multiprocessados não precisam trabalhar com um *clock* elevado, pois o que faz os sistemas multiprocessados serem tão poderosos é o trabalho coletivo, com a disponibilidade de mais de uma CPU é possível executar várias tarefas de forma simultâneas, embora isso não tenha um grande impacto para usuários comuns para os quais o pseudoparalelismo é suficiente. Foi de um importante ganho para a comunidade científica.

Mas com a velocidade o paralelismo também traz problemas que precisam ser levados em consideração na execução de tarefas paralelas, por exemplo, é preciso garantir que uma CPU não interfira na memória utilizada por outra (TANENBAUM; BOS, 2014).

2.1.3 Porque e quando usar paralelismo

O uso do paralelismo pode representar um ganho de desempenho em diversas aplicações. Mas é importante lembrar que o paralelismo não é uma solução para todos os problemas da computação.

É preciso ter uma estratégia para encontrar estes problemas que possam ser resolvidos usando o paralelismo. Um requisito principal para o poder paralelizar um algoritmo é que ele possa ser decomposto em vários subproblemas. Desta forma, cada CPU pode trabalhar com um subproblema. O paralelismo pode fazer uma alusão a técnica de dividir para conquistar.

Também há os problemas que embora sejam paralelizáveis provavelmente não haverá grandes ganhos. Suponha que um cálculo pode ser decomposto em vários subproblemas, mas são necessários valores previamente calculados para calcular o próximo. Isso indica que mesmo que os subproblemas sejam distribuídos entre CPUs a execução não terá ganhos significativos se comparada a execução sequencial (RAUBER; RNGER, 2013).

2.2 Arquitetura de Computadores

> *Hundreds of different kinds of computers have been designed and built during the evolution of the modern digital computer. Most have been long forgotten, but a few have had a significant impact on modern ideas. (TANENBAUM, 2005, pp. 13)*

Existem uma infinidade de aparelhos que podem ser chamados de um computador. Desde chips de placas únicas medindo alguns centímetros à *mainframes* ocupando salas inteiras, igualmente são denominados computadores. A razão para esse nome comum é simples: para ser denominado como um computador é necessário preencher apenas alguns requisitos básicos. Muitos dos computadores existentes são baseados em arquitetura clássicas, principalmente a

arquitetura de John Von Neumann. Dentre esses requisitos estão: Memória capaz de armazenar (não necessariamente de forma permanente) os dados, uma unidade lógica aritmética responsável pela execução de operações básicas, uma unidade capaz de controlar dados e uma forma de movimento de dados entre as três unidades, esse requisitos são definidos pela arquitetura de computadores utilizadas.

Embora a classificação de o que é um computador seja muito simples ao longo da história da computação, a forma que esses computadores são construídos variam. É a partir dessa variação que surge o termo Arquitetura de Computadores, uma arquitetura de computador refere-se a como os atributos de um processador está disponível para um desenvolvedor. Cada arquitetura define como os componentes funcionarão em determinado sistema computacional, funções como o tamanho da palavra no processador, o endereçamento de memória, a representação de dados, a maneira que os dados são movidos entre as unidades, entre outros. Tudo isso que a arquitetura define muda o conceito de desenvolvimento de um programa em uma determinada máquina.

A arquitetura define de que forma os programas serão desenvolvidos, como os primeiros computadores usavam arquiteturas diferente e então os programas funcionavam apenas em um tipo de computador e os programadores muitas das vezes programavam apenas um tipo de máquina. Como uma resposta a essa limitação foi então desenvolvido o conceito de família de computadores, em uma família de computadores todos os computadores compartilham uma mesma arquitetura embora contém com componentes diferentes, podendo ser mais ou menos potentes. O grande trunfo da família de computadores é que desde que a arquitetura seja a mesma, não importa as mudanças de *hardwares*, o *software* implementado para essa arquitetura deve funcionar em qualquer computador.

Embora compartilhem a mesma arquitetura é importante ter ciência de que os computadores não necessariamente compartilhem a mesma organização. A Organização de Computadores é a forma como as funcionalidades serão implementadas: sinais de controle, interfaces entre os periféricos, tecnologia e tipo de memória (TANENBAUM, 2005; STALLINGS, 2015).

2.2.1 Requisitos de Arquiteturas

Todas as arquiteturas buscam prover uma série de requisitos.

2.2.1.1 Requisitos Gerais

- Processar maior quantidade de dados
- Processamento em menor tempo
- Computação no menor espaço
- Menor consumo de energia

2.2.1.2 Para sistemas embarcados

- Baixo consumo
- Dimensão física reduzida
- Baixo custo
- Capacidade Computacional compatível com a aplicação

2.2.2 Conjunto de instruções: RISC vs CISC

Para Stallings (2015), um conjunto de instruções é *A coleção de diferentes instruções que o processador pode executar é referido como um conjunto de instruções.*, em tradução livre. Uma instrução é a forma que a arquitetura define como cada operação de um processador será chamada em linguagem de máquina. Elas são passadas em forma de códigos e cada código é uma sequência de dados são passados de maneira que o processador entenda o que fazer, no código da instrução cada parte indica uma coisa para o processador. É indicado no código de operação qual instrução o processador deve executar, também são passados os operandos que contém os dados, o operando que deve guardar o resultado e por fim a referência para a próxima instrução.

Algumas operações como as operações aritméticas básicas estão disponíveis em todos os conjuntos de instruções, mas algumas instruções complementares não são consenso em todas arquiteturas. Como visto em Tanenbaum (2005) a partir da década de 60 o *design* das arquiteturas ficou polarizado entre duas correntes diferentes, as arquiteturas RISC ou arquiteturas CISC.

As arquiteturas RISC (sigla em inglês: *Reduced Instruction Set Computer*, traduzida como "Computador com um conjunto reduzido de instruções"), baseiam-se em um conjunto simples e limitados de instruções que levam a mesma quantidade aproximada de tempo para serem processadas. Os computadores que usam essas arquiteturas são computadores que necessitam do processador para um conjunto reduzido de funções, por isso as arquiteturas RISC estão mais presentes em computadores para sistemas embarcados.

Em oposição as arquiteturas CISC (do inglês : *Complex Instruction Set Computer*, traduzida como "Computador com um conjunto complexo de instruções. Nesse lado oposto do debate os computadores possuem uma grande quantidade de instruções, das quais muitas delas não são utilizadas. Essas instruções tornam mais singles a geração de código de máquina, pois muitas das instruções contidas nos programas já possuem equivalência direta com uma instrução do processador, no entanto há perda de desempenho.

2.2.3 Arquitetura x86

A arquitetura x86 é assim denominada porque os nomes dos primeiros modelos dessa família tinham números acabados com o número 86, com um conjunto de instruções baseado no Intel 8086. A arquitetura x86 acabou se tornando um padrão para os computadores pessoais sendo seu conjunto de instruções implementado por várias marcas de computadores. Essa padronização torna possível a compartibilidade entre os programas entres os computadores das mais diferentes marcas existentes (TANENBAUM, 2005; STALLINGS, 2015).

2.2.3.1 Principais características da arquitetura x86

- Conjunto de instruções CISC estendido
- Acesso a memoria permitido por todos tamanhos de palavra suportados
- Tamanho de inteiros de 16, 32 ou 64 bits
- 8 registradores de proposito geral de 32 bits cada
- 6 registradores de segmento
- 1 registrador de flag
- 1 registrador de execução

2.2.4 Arquitetura ARM

As arquiteturas ARM (do inglês: *Advanced Risk Machine*) são um conjunto de arquiteturas atualmente desenvolvidas pela empresa britânica ARM Holdings. No entanto as primeiras arquiteturas foram desenvolvidas pela, também britânica, Acorn em 1983. Onde a responsabilidade pelo desenvolvimento do conjunto de instruções ficou com Sophie Wilson (LIMA; MORENO; DIAS, 2016).

Ainda na década de 1980 houve um interesse por parte da Apple em utilizar processadores ARM, no entando algumas mudanças no projeto foram necessárias. Assim em novembro de 1990, em uma parceria da Acorn, Apple e VLSI surge a empresa ARM Holdings.

A produção de chips nunca foi o objetivo da ARM Holdings, por esse motivo a empresa costuma licenciar sua propriedade intelectual para várias outras empresas. Essa característica leva as arquiteturas ARM ao topo das arquiteturas quanto ao número de chips produzidos, contando com bilhões de chips.[2]

Segundo o manual de referência (SEAL, 2000), as arquiteturas ARM suportam um amplo espectro de pontos de performance. A simplicidade dos processadores levam a uma pequena

[2] Disponível em: https://community.arm.com/iot/b/blog/posts/arm-from-zero-to-billions-in-25-short-years

implementação. Os processadores ARM são baseados em um conjunto de instruções RISC e, por isso, o conjunto de instruções é simplificado. Por conta do conjunto de instruções RISC que permitem essa implementação simples, essa arquitetura é desejada por fabricantes de processadores para dispositivos portáteis. Os registradores internos são usados para processamento de dados, onde a CPU lê os dados da memória principal e armazena nos registradores internos (LIMA; MORENO; DIAS, 2016).

2.2.4.1 Principais características da arquitetura ARM

- Conjunto de instruções RISC
- Instruções de tamanho fixo de 32 bits ou 64 bits
- 16 registadores de proposito geral de 32 bits cada
- Tamanho do núcleo reduzido
- Baixo consumo de energia

2.3 Sistemas Distribuídos

O conceito de Sistema Distribuído não é universal, com vários autores definindo de formas diferente. Para Coulouris, Dollimore e Kindberg (2005) um Sistema Distribuído é *aquele no qual os componentes de hardware ou software, localizados em computadores interligados em rede, se comunicam e coordenam suas ações apenas enviando mensagens entre si*. Já para Tanenbaum e Steen (2006) é *uma coleção de computadores independentes que aparecem para o usuário como um sistema único*.

Embora as muitas definições sejam diferentes, elas convergem no sentido de explicar o que é um Sistema Distribuído. O ponto inicial para o surgimento de Sistemas Distribuídos foi a popularização das Redes de Computadores, sobretudo a *Internet*. Com os computadores interligados pelas redes, os sistemas deixaram de trabalhar de forma única e isolada para funcionar com base na comunicação de partes que estão distantes.

Nesse contexto, onde as estão informações disponíveis e são trocadas por Redes de Computadores, o estudo de Sistema Distribuído como um ramo da ciência da computação se desenvolveu e independente das várias definições. O fato é que um Sistema Distribuído é um modelo de computadores que estão conectados em rede e com base na troca de mensagens executam uma ou mais tarefas de maneira conjunta. Onde os componentes de *hardware* e *software* interagem com um objetivo em comum. Três características são definidas para a criação de um sistema distribuído, são elas: concorrência, a falta de um relógio global e a falha independente de componentes (COULOURIS; DOLLIMORE; KINDBERG, 2005; TANENBAUM; STEEN, 2006).

A concorrência está ligada a dinâmica das Redes de Computadores, onde as máquinas parecem está isoladas e fazem suas tarefas sem aparente dependência uma da outra, mas muitas das vezes essas máquinas dependem de um recurso disponibilizado pela rede, assim o recurso é compartilhado entre diferentes máquinas. Cabe ao Sistema Distribuído gerenciar a dinâmica de distribuição deste recurso.

Já a falta de um relógio global está ligada a necessidade dos Sistemas Distribuídos de uma noção de tempo enquanto cooperam e coordenam suas ações durante as trocas de mensagens. Entretanto existem limites para a noção de tempo entre computadores ligados em uma rede. Para isso é parte fundamental dos Sistemas Distribuídos a sincronização das trocas de mensagens.

A falha independente dos componentes dos componentes está ligada a possibilidade de qualquer sistema computacional entrar em colapso. O que diferencia a falha de um sistema comum para a falha em um Sistema Distribuído é que a falha em um único componente não deve fazer com que o Sistema Distribuído pare de funcionar. De forma separada os componentes do sistema devem detectar que um componente falhou e devem trabalhar de forma que o sistema continue funcional mesmo sem um dos seus componentes.

Muitas aplicações podem ser construídas como um Sistema Distribuído, mas como definem Tanenbaum e Steen (2006) não é porque somos capazes de criar um Sistema Distribuído que devemos simplesmente criá-los. É preciso que o sistema que está sendo criado tenha um objetivo para que o esforço de criar um Sistema Distribuído seja recompensado. Tanenbaum e Steen (2006) também definem que um Sistema Distribuído deve fazer com que seus recursos sejam facilmente acessados, deve razoavelmente esconder o fato que os recursos estão distribuídos pela rede, deve ser aberto e deve ser escalável.

Dentre as múltiplas aplicações de Sistema Distribuído a mais importante é a sua aplicação na resolução de problemas que usam paralelismo (Ver Seção 2.1) e a utilização de Sistemas Distribuídos na construção de *clusters* (ver Seção 2.4).

2.3.1 Principais Características de um Sistema Distribuído

- Heterogeneidade: O sistema deve permitir na sua construção uma variedade de linguagens de programação,*hardwares*, redese Sistemas Operacionais. Os quais devem ser homogeneizados pela junção dos protocolos de internet e dos *middlewares*.
- Sistema Aberto: O sistema deve ser expansível, para isso as interfaces dos componentes devem ser públicas.
- Segurança: Deve ser usada criptografia para manter sigilo das mensagens trocadas na rede.
- Escalabilidade: Os sistemas devem permitir que novos componentes sejam adicionados.
- Transparência: Alguns aspectos da distribuição devem está invisíveis para o programador.

2.3.2 Arquiteturas de Sistemas Distribuídos

- Cliente-Servidor: Esse é o modelo mais clássico de arquitetura, nele uma máquina funciona como Servidor e todas as outras são Clientes. Os clientes enviam mensagens de requisições ao Servidor e o mesmo responde as solicitações. Casos os Clientes precisem se comunicar entre si, o Servidor precisa funcionar como uma ponte.
- *3-tier*: Nessa arquitetura a parte lógica do sistema é movida para uma camada intermediaria. Os Clientes contém a camada de apresentação e os Servidores armazenam os dados.
- *n-tier*: Nas arquiteturas n-tier qualquer parte do sistema que faz requisições é chamado de Cliente, enquanto as partes que respondem as requisições são Servidores.
- *peer-to-peer*: Não existe definições de Clientes ou Servidores, todas as partes do sistema são igualmente responsáveis por realizar ou responder requisições.

2.4 Cluster

Um *cluster* é um sistema homogêneo de dois ou mais computadores capazes de realizar uma tarefa objetivo , onde cada computador é conectado por rede (TANENBAUM, 2005). Geralmente essa rede é uma rede local, conectando os diversos computadores que são chamados de nós do *cluster*. Geralmente os nós têm um mesmo *hardware* e usam o mesmo Sistema Operacional, mas isto não é uma regra.

O conceito de *cluster* assemelha-se ao conceito de Sistema Distribuído. Assim como os Sistemas Distribuídos (ver Seção 2.3), um *cluster* deve ser visto como um sistema homogêneo que realiza uma tarefa especifica, além da necessidade de continuar funcionando mesmo sem um dos seus nós. Geralmente os *clusters* são programados e controlados por *softwares*.

Com os primeiros clusters já sendo desenvolvidos na década de 60. A implementação de um *cluster* geralmente é motivada pela necessidade de performance. A utilização de um *cluster* representa um potencial ganho de desempenho, pois os nós trabalhando em conjunto propiciam paralelismo (ver Seção 2.1).

Clusters geralmente são organizados principalmente duas maneiras distintas: Os *clusters* centralizados, onde os computadores usados estão reunidos em uma mesma sala, geralmente em um *hack*, e os *clusters* descentralizados, onde os computadores estão em salas diferentes ou até mesmo em prédios diferentes. Como a conexão entre os nós é realizada via rede, é importante observar que a distância entre os nós pode representar um fator a ser considerado durante a construção de um *cluster*.

A Figura 1 mostra um modelo de *cluster* conectado em rede. Cada computador conectado ao *cluster*, chamado de nó, conta com memória e CPU própria. Cada nó realiza uma tarefa, que é enviada via rede e os dados são armazenados e processados na memória de cada nó.

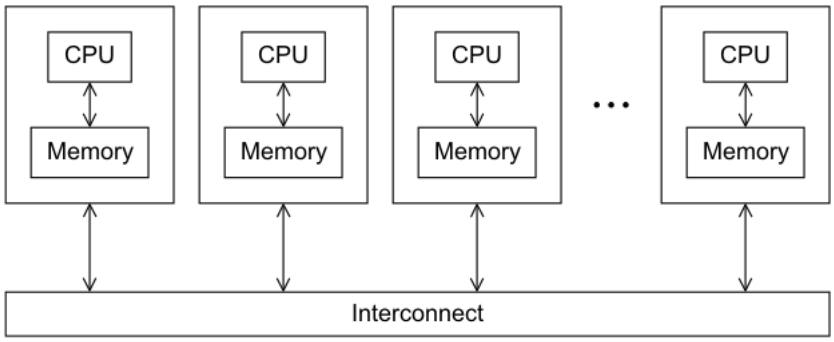

FIGURE 2.4
A distributed-memory system

Figura 1 – Memória distribuída em um *cluster*

Fonte – (PACHECO, 2011)

No modelo de construção de um *cluster* são utilizados geralmente vários computadores (pelo menos dois) onde um computador é chamado de servidor ou mestre, e os restantes são chamados de escravos. Os computadores de um *cluster* geralmente utilizam um *hardware* comum, ou seja, todos os nós do *cluster* são computadores idênticos. E os nós do *cluster* são conectados via rede e *software*.

2.4.1 Tipos de Clusters

Existem três principais tipos de *clusters*. São eles:

- Cluster de Alto Desempenho: Funciona utilizando o máximo disponível se processamento, sendo altamente usado em aplicações de Computação de Alto Desempenho (ver Seção 2.5).
- Cluster de Alta Disponibilidade: São *clusters* que sempre devem estar ativos, para isso é importante que detectem erros e adaptem-se a falhas.
- Cluster de Balanceamento de Carga: Deve equilibrar a distribuição de processamento entre os nós. Para tanto, é preciso o constante monitoramento da comunicação entre os diferentes nós. Caso haja algum erro, esses *clusters* geralmente interrompem seu funcionamento.

2.5 Computação de Alto Desempenho

> *The ENIAC could perform 300 operations per second, easily 1000 times faster than any calculator before it, yet people were not satisfied with it. We now have machines millions of times faster than the ENIAC and still there is a demand for yet more horsepower (TANENBAUM; BOS, 2014, pp. 517).*

Mesmo com o aumento exponencial de desempenho que os computadores tiveram nas últimas seis décadas, o aumento não foi suficiente para acompanhar a necessidade de processamento. Embora o processamento disponível em computadores pessoais seja suficiente para a realização de todas as tarefas de um usuário comum, não é o suficiente para os problemas enfrentados por diversas áreas do conhecimento. Nesse contexto surge a Computação de Alto Desempenho (PACHECO, 2011).

O uso da Computação Científica não é exclusivo da Ciência da Computação. Biólogos necessitam de supercomputadores para sequenciar DNA, Químicos e Físicos representam seus modelos teóricos em computadores, múltiplas engenharias realizam seus cálculos em computadores. A Computação de Alto Desempenho é a área que busca tornar experimentos dessas e de outras áreas possíveis, estudando forma de realizar os cálculos e outra operações realizada sobre os dados mais rápidos. Muitas das técnicas de Computação de Alto Desempenho estão relacionadas ao uso de paralelismo (ver Seção 2.1) e o uso de *clusters* (ver Seção 2.4).

Existem diversas técnicas para mensurar o que é uma Computação de Alto Desempenho. A performance de um sistema computacional é um dos pontos mais importante quando precisa-se medir a eficiência, enquanto os usuário comuns estão interessados no tempo de reposta, isso é, o tempo que demora entre o acionamento de um programa e o término de sua execução. A Computação de Alto Desempenho está mais interessada em medir os *throughputs*, que é a média de trabalhos que cada unidade de processamento pode executar em uma unidade de tempo.

As performances dos sistemas computacionais utilizados em computação de Alto Desempenho são muitas vezes medidas em MIPS (*Millions Instructions Per Second*). O cálculo é feito exclusivamente no número de instruções não importando que algumas instruções demorem mais do que outras. Por esse motivo, a taxa de MIPS não significa necessariamente o tempo de execução de um programa. Além disso, o uso de programas de *benchmark* são usados constantemente para definir a performance. Os *benchmarks* geralmente avaliam a performance de componentes de *hardware*, por exemplo, a performance da operação de ponto flutuante (RAUBER; RNGER, 2013).

Existem três principais tipos de *benchmarks*. São eles:

- Sintéticos: Geralmente executam pequenos programas artificiais com uma mistura de operações que fazem parte de um programa real. Comumente não executam operações em um conjunto significativo de dados.
- De Kernel: Executam pequenas, mas significantes parte de um programa real. Essas pequenas partes são responsáveis pela maioria do tempo de processamento.
- De Aplicação Real: Executam alguns programas inteiros.

2.6 *Apache Hadoop*

O *Apache Hadoop* é um *framework* escrito principalmente em Java que permite o desenvolvimento de processamento distribuído de uma grande quantidade de dados em *clusters* usando modelos de programação simples. O *framework* é projetado para ser escalável desde servidores únicos a centenas de máquinas em um *cluster*, oferecendo processamento e armazenamento locais (Apache Software Foundation, 2017).

Embora a maior parte do *framework* seja escrito em Java (também há partes escritas em C e em Shell Script), o código do *Map Reduce* do java é comum. O MapReduce é um modelo de programação distribuída usada para gerar e processar conjuntos muito grandes de dados de forma paralela usando algoritmos distribuídos em um *cluster*.

A chave de utilização do MapReduce está em duas funções. A função Map, que que filtra e ordena os dados a serem processados, enquanto a função Reduce de fato realiza as operações sobre os dados (DEAN; GHEMAWAT, 2008). O MapReduce funciona no Hadoop usando o JobTracker e o TaskTracker. O JobTracker envia dados para os nós que estão ociosos e sabe em quais nós os dados estão armazenados, enquanto o TaskTracker gera uma máquina virtual Java separada em cada nó para impedir que os nós falhem e constantemente enviam seu *status* ao JobTracker.

Dessa maneira é possível através do *Hadoop Streaming*, escrever códigos em qualquer linguagem de programação que implementem o *Map* e o *Reduce*. O núcleo do *Hadoop* é formado principalmente por quatro módulos, todos os módulos do *Hadoop* são desenvolvidos com a fundamental suposição que o *hardware* pode falhar, dessa forma o *Hadoop* deve via *software* lidar automaticamente com essas falhas (BAPPALIGE, 2014).

Os quatro módulos são:

- *Hadoop Common*: Contém as bibliotecas e as ferramentas utilizadas em outros módulos.
- *Hadoop Distributed File System*: É um sistema de arquivos que distribui os arquivos através dos nós do *cluster Hadoop*,provendo uma largura de banda muito alta.
- Hadoop YARN: É a plataforma de gerenciamento de arquivos responsável por gerenciar os recursos dos computadores no *cluster* e usa-los para o agendamento das aplicações de usuário.
- *Hadoop MapReduce*: É a implementação do *Hadoop* para um modelo de programação em larga escala desenvolvido pela Google.

Os dois principais componentes do *Hadoop* são o *Hadoop Distributed File System* (HDFS) e o *Hadoop MapReduce*. ambos são baseados em projetos *open-source* disponibilizados pela Google.

O HDFS é escalável, distribuído e portável, cada nó em um *cluster* tipicamente tem um *namenode* e um agrupamento de *datanodes* forma o HDFS. Cada *datanode* envia blocos de dados através de um protocolo específico para o HDFS. O sistema de arquivos usa a camada de TCP/IP para se comunicar, enquanto os clientes se comunicam por chamadas de procedimento remoto. O sistema armazena arquivos grandes distribuídos através de múltiplas máquinas e a confiabilidade é dada pela replicação de arquivos através dos *hosts*. Também é parte do HDFS um *namenode* secundário, embora o nome secundário confunda fazendo acreditar que o secundário entra em ação caso o primário falhe, esta não é a sua função. O *nomenode* secundário funciona entrando em contado regulamente com o primário e armazenando algumas informações (BAPPALIGE, 2014).

2.7 *OpenMPI*

O OpenMPI é um projeto *open-source* implementado e desenvolvido por uma grande equipe de pesquisadores, desenvolvedores e acadêmicos. É uma implementação de *Message Passing Interface (MPI)*, que combina recursos, experiências e tecnologias existentes para criar a melhor ferramenta de MPI possível (OPENMPI, 2018). O OpenMPI é apenas uma implementação do protocolo que foi escolhida para este trabalho, por contar também com uma implementação para a linguagem Java e está disponível para a plataforma *Raspberry PI*.

O modelo de programação *Message-Passing* é uma abstração da computação paralela em um ambiente distribuído onde cada processador tem uma memória local com acesso exclusivo. Uma memória global não existe e as trocas de dados precisam ser realizadas por uma troca de dados. Para um processador A fazer a troca de dados com um processador B enviando uma mensagem atráves da rede a qual os nós do *cluster* estão conectados. Para garantir a portabilidade nenhuma consideração sobre a rede onde as mensagens são enviadas pode ser feita, apenas pode-se assumir que os processadores consigam se comunicar (RAUBER; RNGER, 2013).

O protocolo MPI funciona com base em tarefas, onde cada tarefa a depender da implementação pode ser um processo ou *threads*. Cada core disponível no processador pode receber uma tarefa. Não existe um modelo de memória global e os dados devem ser enviados por mensagens para cada tarefa. Como não existe uma memória global, o mesmo código fonte é executado em cada processador fazendo as declarações de vvariáveis (LIMA; MORENO; DIAS, 2016). Cada tarefa recebe um número chamado de *RANK* e é a partir desse número que é possível definir com qual tarefa se comunicar.

3

Trabalhos Relacionados

A natureza desse trabalho é a construção de um experimento baseado na comparação de dois algoritmos similares construídos em linguagens diferentes, e tal experimento é realizado em um *cluster* baseado em *Raspberry* PI. Com intuito de estabelecer as bases para construção de tal experimento, foi realizada uma revisão acerca dos trabalhos que fizessem o uso de técnicas já conhecidas de *clustering* utilizadas com uma abordagem diferente, ou seja, voltadas para computadores ARM *Raspberry PI*. O objetivo da revisão feita, foi conhecer quais trabalhos já foram publicados e quais experimentos já foram feitos. Para a obtenção de tais informações foram utilizadas as principais bases de trabalhos científicos da área de computação.

De posse dos resultados encontrados no presente capítulo foi possível definir quais experimentos seriam feitos no presente trabalho e de qual forma representaria algum avanço para a área de Computação de Alto Desempenho.

O processo de pesquisa, a discussão sobre os trabalhos encontrados e, por fim, os objetivos delineados a partir dos resultados encontrados serão descritos nas Seções a seguir.

3.1 Revisão Sistemática

Seguindo o procedimento indicado em Kitchenham (2004), esse capítulo faz uma pesquisa em bases cientificas em busca de trabalhos a partir de *strings* de busca, depois seleciona e qualifica os artigos de acordo com a relevância. Após um refinamento entre todos os artigos encontrados, apenas os selecionados farão parte da discussão do restante do capítulo.

As pesquisas na bases foram realizadas no período que vai de dezembro de 2017 a janeiro de 2018 e para eliminar a restrição de download das bases foi utilizado o portal Capes.

3.1.1 Bases Utilizadas

Para a realização da pesquisa foram utilizadas as seguintes bases:

- ACM Digital Library
- IEEE Xplorer
- Science Direct
- Spring Link

As seguintes bases não foram utilizadas por não disporem de um mecanismo avançado de busca.

- Google Scholar
- BDBComp

3.1.2 Termos de busca e String de busca

Para tornar possível a busca de artigos foram formulados seis termos de busca, são eles:

1. Cluster
2. Clustering
3. Raspberry PI
4. High Perfomace Computing
5. Parallel Processing
6. Performace Evaluation

A partir da combinação deste termos foi cunhada uma *string* de busca genérica, que foi utilizada igualmente em todas as bases.

- ((cluster OR clustering) AND Raspberry PI OR Raspberry PI AND(High Perfomace Computing OR Perfomace Evaluation))

3.1.3 Critério de seleção e relevância dos artigos encontrados

Para delimitação dos artigos que seriam selecionados, foi realizada a leitura do resumo e conclusão de todos artigos encontrados nas bases de busca. A partir daí foram selecionados artigos que apresentassem de forma teórica e/ou prática a utilização de *cluster* para o desenvolvimento de Computação de Alto Desempenho.

Quanto aos critérios de exclusão, foram excluídos artigos encontrados em mais de uma base. Também não foram utilizados aqueles que não foi possível o *download* do texto integral ou não estivessem no escopo da pesquisa, foi considerado fora do escopo qualquer artigo que abordasse uso de *Raspberry PI* sem relação com Computação de Alto Desempenho ou Computação de Alto Desempenho sem relação com *Raspberry*.

Por fim a avaliação da relevância dos trabalhos selecionados foi dada a partir da leitura completa dos mesmos.

Acabado o processo de busca, inicia-se o processo de discussão acerca dos resultados encontrados, buscou-se nesse capítulo detectar quais experimentos parecidos já foram realizados e por fim detectar algo ainda inexplorado.

3.2 Discussão acerca dos trabalhos relacionados

O lançamento do primeiro modelo Raspberry PI foi em 2012[1], e o primeiro trabalho encontrado nas bases pesquisadas relacionado ao uso como um sistema distribuído data de 2013. Em Tso et al. (2013) foi formulado a *"The Glasgow Raspberry PI Cloud"* onde seus autores propuseram um modelo de nuvem escalável baseado no Raspberry PI. As seguintes subseções descreverão os objetivos, metodologia e conclusão de cada um dos artigos selecionados. Findada a descrição dos artigos encontra-se a Tabela 1 fazendo uma comparação entre os artigos.

3.2.1 *The Glasgow Raspberry Pi Cloud: A Scale Model for Cloud Computing Infrastructures*

Tso et al. (2013) argumentam que a computação em nuvem é um novo paradigma de processamento baseado na infraestrutura do *pay-as-you-go*. Eles argumentam que existem um número de problemas que precisam ser discutidos acerca do modelo tradicional de nuvens em *Data Centers*, um deles é o custo. Também salientam que mesmo as pesquisas em simulação de nuvem tenha grande importância para o desenvolvimento de algumas áreas da computação em nuvem, a simulação é falha para muitas coisas sobretudo, na simulação do tráfego na nuvem. A partir disso eles propõem uma nova alternativa na pesquisa de nuvem, o que eles chamam de *CloudPi*. Um modelo de nuvem escalável composta por 56 *Raspberry PI*.

[1] Data disponível no site oficial: www.raspberrypi.org

A arquitetura do sistema do *Cloud PI* foi feita usando os 56 nós conectados por uma arvore topológica de múltiplas raízes, essa organização foi escolhida com intuito de refletir a organização dos *Data Centers*. Cada parte do sistema roda um *Raspbian* em um *SD-Card* de 16Gb e acima da camada do Sistema Operacional, são usados *Containers* para que seja possível a virtualização. Para conseguir virtualizar no *Raspberry* modelos tradicionais de virtualização como o *VMware* não são possíveis por conta da pouca quantidade de RAM disponível, por isso na construção da nuvem foi utilizados *containers* que usam apenas 30MB de RAM.

Os autores argumentam que com a *CloudPi* eles podem testar qualquer aspecto de uma nuvem convencional. Também afirmam que conhecendo a maneira que a nuvem é virtualizada, eles podem testar novos algoritmos e avaliar empiricamente qualquer outro aspecto das outras camadas. Por exemplo, é possível medir o congestionamento da rede na nuvem.

O custo estimado por Tso et al. (2013) para a construção da *CloudPi* foi 1960 dólares enquanto uma nuvem de 56 computadores x86 custaria 112.000. Por fim, os autores concluem que o principal valor da construção da *CloudPi* é tornar a experiência na gestão de *Data Centers* mais real do que os simuladores.

3.2.2 *A Container-based Edge Cloud PaaS Architecture based on Raspberry Pi Clusters*

Em (PAHL et al., 2016), também é abordada a forma de construção de *cluster* de *Raspberry PI* como opção para a construção de uma nuvem. No entanto, a diferença está no fato que, segundo os autores, as nuvens não estão mais centralizadas em *Data Centers* e sim distribuídas, essas nuvens são chamadas de *edge clouds*. Esta abordagem está relacionada a necessidade de conectar *Data Centers* com aparelhos pequenos conectados a internet. No entanto existe uma barreira que é a virtualização, que para ser usada custa uma grande quantidade de memória RAM, que muitas das vezes, esses aparelhos não dispõem. Por isso é preciso criar *containers* menos custosos quanto ao consumo de memória RAM. O *Raspberry Pi* é uma alternativa para testar esses *containers*.

Os autores definem a arquitetura para essas *edge clouds*, tais como a necessidade dos *containers*, e a forma de gerenciá-los. Após a construção de uma nuvem composta por 300 nós, é analisado o custo benefício, baseados em fatores como custo, consumo de energia e robustez.

3.2.3 *Modelling Low Power Compute Clusters for Cloud Simulation*

Kecskemeti, Hajji e Tso (2017) discutem no seu trabalho a possibilidade de construir um *Data Center* baseado em computadores ARMs, pensando, além do baixo custo de comprar essas placas em quantidade, no baixo consumo de energia. Inspirados pela *CloudPi* definida em (TSO et al., 2013), comparam o desempenho de uma *Raspberry Pi* com uma nuvem formada por computadores x86.

O artigo trata as limitações causadas na implementação de nuvens em computadores ARM por conta da maioria das ferramentas terem sido projetada para arquitetura x86, por isso os autores precisaram modificar os simuladores existentes para que pudessem ser utilizados em computadores ARM. Foram feitos testes em cenários reais e simulados.

O artigo defende que os *Data Centers* gastam muita energia e que o uso de computadores com baixo de consumo como o *Raspberry Pi* está sendo ignorado no desenvolvimento dos simuladores. E iniciativas como a de criação de simuladores que funcionem em arquiteturas ARM possam mudar essa perspectiva.

3.2.4 Comparison of Load Balancing Methods for Raspberry–Pi Clustered Embedded Web Servers

Maduranga e Ragel (2016) abordadam a criação de servidores embarcados. Para isso, os autores criam um *cluster* com três *Raspberry Pi*. Um deles funciona como mestre e os outros dois como escravos. O objetivo do trabalho é verificar o desempenho e escalabilidade do *cluster*. Usando o *Apache Web Server*, é medido o desempenho de quatro algoritmos de balanceamento de rede em um servidor web embarcado.

Os quatro algoritmos são:

- *By_Request*: O servo com maior *lbstatus* é selecionado, então o valor do *lbstatus* é decrementado do *lbfactor*. Consequentemente a soma de todos *lbstatus* e o *request* são distribuídos.
- *By_Traffic*: Funciona de maneira bastante a o *By_Request*, no entanto o balanceamento não é feito pelo tráfego na rede.
- *By_Busyness*: O número de *requests* em cada servo é armazenado. Quando chega um novo *request*, o servo com menor número fica com o *request*.
- *By_Heartbeat*: Cada *request* é entregue ao servo mais ocioso.

O cálculo para comparação foi feito com base nas sequintes métricas:

- Transações: O total de transações é dado pelo número de usuário multiplicado pela quantidade de *requests* permitida por usuário.
- Tempo Gasto: O tempo decorrido entre o início e o termino do experimento.
- Dados Transferidos: A soma de todos o dados de usuário como resposta a todos os *requests* feitos.
- Taxa de Transações: O número de transações por segundo.

- Concorrência: A média de transições simultaneas de um usuário pela média de todos os usuários.
- Transações Bem Sucedidas: A quantidade de transações que são consideradas bem sucedidas, ou seja, aquelas que o server retorna um número menor de 400.

Os testes foram realizados com 10, 100 e 1000 usuários. No aumento de 10 para 100 só foram apresentadas mudanças significativas nas taxas de transição e concorrência. Mas na mudança de 100 para 1000 todos parâmetros apresentam mudanças significativas. O algoritmo *by request* foi o que se saiu melhor nos testes.

3.2.5 *Study of Raspberry Pi 2 Quad-core Cortex-A7 CPU Cluster as a Mini Supercomputer*

O trabalho de Mappuji et al. (2016) assemelha-se com o objetivo deste trabalho, porque os autores tocam no ponto do alto custo e grande consumo de energia na Computação de Alto Desempenho usando os supercomputadores convencionais. Baseado em estudo anterior (MOORE, 2014) já se sabia que um *cluster* de quatro nós de um *Raspberry PI single core* e com 0.7 GHz de *clock* atinge 846 MFLOPS usando o *benchmark* HPL. Dessa maneira o objetivo do estudo é identificar o desempenho de *cluster* de 4 nós (*quad- core*) quando o *clock* é aumentado para 0.9Ghz.

Os estudos foram realizados com base em duas comparações: consumo de energia versus quantidade de nós e performance computacional versus quantidade de nós. Na análise do consumo de energia foi identificado que todos os nós consumem 1.19W quando está ocioso e 1.49W quando está realizando processamento. Na análise de desempenho ficou constatado que o desempenho com um nó é de 150 MFLOPS e após adição dos outros três nós o resultado final é de 850 MFLOPS nos computadores *quad-cores*.

Baseado na normalização da curva de performance com o número de nós, concluem que o número ideal são 4 nós, pois há um ganho significativo entre as curvas de 2 e 4 nós, aliado com o fato que 2 nós são muito pouco para um *cluster* de Alto Desempenho. Também fica claro que quantidade de *cores* em cada *Raspberry* não representa ganhos significativos, pois o *cluster single-core* obteve 836 MFLOPS no *benchmark* enquanto o *quad-core* atigiu 850.

3.2.6 *Performance of a Low Cost Hadoop Cluster for Image Analysis in Cloud Robotics Environment*

Inspirados pelo rápido desenvolvimento das *cloud robotics*, cujo o mercado popularizou sob uso dos *drones*, permitindo aplicações em diferente modalidades. Este artigo aborda seu uso na captação de imagens aéreas Qureshi et al. (2016) procuram estudar a performance de um *cluster* baseado em *Raspberry PI* aliado a essas *clouds*. A razão para essa pesquisa é que os

drones, controlados por controle remoto, usam de forma principal a sua câmera para permitir sua navegabilidade. As imagens capturadas pela câmera precisam ser processadas para permitir o uso de *drones* em aplicações reais, tais como: identificação de objetos suspeitos.

A análise dessas imagens requer um grande processamento,pois cada operação em uma imagem requem um calculo na matriz de pixels desta imagem (PEDRINI; SCHWARTZ, 2008), o processamento no *drone* é limitado e este processamento seria lento. Por isso, uma das estratégias empregadas para contornar esse problema é a transferência dos dados capturados para nuvens, que realizam o processamento e transferem o resultado de volta para o *drone*. Com isso, o objetivo do trabalho é analisar a performance de um *cluster* construído com o *framework Hadoop* (ver Seção 2.6) para análise de imagens. No entanto o *Hadoop* apresenta algumas limitações no uso de imagens, como:

- Não existe uma interface para ler ou escrever imagens.
- A maioria das aplicações para processamento de imagem ou video não são compatíveis com o *Hadoop*.
- A performance dos algoritmos de análise de imagem deixa muito a desejar por conta da biblioteca HIPI.

O *cluster* foi construído com vinte *Raspberry Pi 2* que foram conectados através de uma estrutura de rede baseada em topologia de estrela, conectado em 24 portas de *gigabit* por segundo. E a versão instalada do *Hadoop* foi a 2.6, com configurações que permitem um maior uso de memória, garantindo o melhor desempenho. Mesmo com o *Hadoop* provendo inúmeras interfaces para leitura e escrita de texto, o mesmo não é verdade para imagens. Portanto, foram necessárias alterações na biblioteca HIPI que permitissem a realização dos testes.

Os testes foram realizados com base em três métricas para a avaliação de desempenho: performance computacional, performance da entrada e saída de texto e performance para entrada e saída de imagens. Os experimentos foram realizados com base em três configurações diferentes do *cluster*, e o tempo de execução foi comparado em cada uma das configurações. Em todas as configurações os *Raspberry PI* eram escravos enquanto um computador *Intel I7* era o mestre.

As três configurações foram:

- Configuração 1: Cada nó configurado com o *clock* padrão de 700 MHz, enquanto o *Hadoop* estava configurado para o HDFS ter o tamanho padrão de 128MB e o *MapReduce* e *YARN* com alocação máxima de memória padrão de 426MB.
- Configuração 2: Cada nó configurado com o *clock* de 1000 MHz, enquanto o *Hadoop* estava configurado para o HDFS ter o tamanho padrão de 128MB e o *MapReduce* e *YARN* com alocação máxima de memória padrão de 852MB.

- Configuração 3: Para fins de *benchmark* foi configurado um *cluster* composto por cinco computadores *Intel I7* de 3 GB de RAM, rodando nós em máquinas virtuais e conectados ao mestre de forma semelhante a dos *Raspberry PI*.

Como *benchmark* os autores usaram três critérios, que foram:

- Critério A: Execução de um programa de calculo de PI usando um algoritmo quase Monte Carlo para encontrar **m** dígitos binários da constante PI.
- Critério B: Execução do *WordCount* para verificar a ocorrência de palavras em arquivos de 3MB, 30MB e 300MB.
- Critério C: Analisar o tempo de execução para imagens JPEG de alta resolução usando a biblioteca HIPI..

Com base no critério A , a configuração 2 foi melhor que a 1, mas foi inferior a configuração 3. No critério 2 a configuração 2 é melhor em 50% para arquivos de 3MB e 27% e 19% para os arquivos de 30MB e 300MB respectivamente. A análise do critério C mostrou que a configuração 3 atinge um tempo considerado aceitável e que para configurações 1 e 2 existe uma diferença considerável de desempenho quando a biblioteca HIPI é modificada em relação a original. Com base nos critérios B e C foi constatado que o *cluster Raspberry PI* perde em desempenho em relação ao uso de máquinas virtuais, quando o conjunto de dados é grande.

3.2.7 Avaliação de Cluster Raspberry Pi para Execução de Aplicações de Análise de Imagens Microscópicas Médicas

Á medicina é uma das muitas Áreas do Conhecimento que necessitam da Computação de Alto Desempenho para realizar suas tarefas. Uma dessas aplicações é a análise de imagens microscópias para o reconhecimento de câncer cerebral. Dessa maneira, o trabalho de Ramos, Ralha e Teodoro (2016) trata da construção de um *cluster* formado por 16 *Raspberry PI* como uma alternativa para um ganho de desempenho com um baixo custo de produção.

Além do *cluster Raspberry PI* foram usados outros dois computadores convencionais *core2duo* e I7 respectivamente. Para fins de análise comparativa entre os três foram levados em consideração fatores como o custo, consumo de energia e desempenho. Os 16 nós do *cluster* foram configurados com o Sistema Operacional *Raspbian* e conectados por um *switch* 24 portas de 100MB. A comunicação e gerenciamento dos processos foi realizada usando MPI (ver Seção 2.7).

Para a análise das imagens microscópicas é necessária a realização de algumas operações, que são caracterizadas quanto a quantidade de acesso de dado e a intensidade computacional. As operações de acesso aos dados podem ser caracterizadas por dois tipos, as regulares e as

irregulares. São ditas como regulares quando fazem acessos contínuos na imagem e irregulares quando fazem acesso a pontos espalhados pela imagem. Essa fase de analise das imagens é chamada de segmentação e foi implementada em C++ usando MPI e a execução é feita utilizando o modelo Mestre/Escravo, onde o mestre armazena a informação sobre as imagens que precisam ser processadas e assinala as mesmas para execução nos escravos sob demanda. Cada escravo então lê a imagem a ser processada e invoca o estágio de segmentação que é executado sequencialmente.

Os experimentos foram realizados com 512 imagens de 1K x 1K pixeis totalizando um tamanho de 6GB. O primeiro fator a ser analisado foi a velocidade de acesso aos dados, que foi uma preocupação inicial dos autores pois o *Raspberry Pi* usa cartões *MicroSD* que são mais lentos que os HDs SATA, além disso o processador trabalha em uma velocidade de *clock* menor. Ficou constatado que o tempo de leitura do *Raspberry PI* é consideravelmente inferior que o de computadores com HD SATA e que isso pode ser um gargalo na execução de problemas com operações de entrada e saída mais intensivas.

Quanto a análise de desempenho por tipo de operação, o *Raspberry PI* foi inferior as plataformas *Core2Duo* e I7, nas operações regulares pois tem seu tempo de acesso a memória mais lento e uma memória cache menor, ficando assim em desvantagem. No entanto, nas operações irregulares, como nenhuma arquitetura otimiza o acesso aleatório e a memória cache não exerce influência o *Raspberry PI* levou vantagem.

Com relação à escalabilidade ficou comprovado que a execução paralela é 47x mais rápida que a execução sequencial, levando a uma eficiência de quase 75%, quando os quatro núcleos do *Raspberry PI* são utilizados, a eficiência é de 80%. É verificado que a perda de eficiência não está relacionada ao uso de múltiplas máquinas e sim de múltiplos núcleos, essa perca de eficiência é causada pela quantidade de acesso a memória. Como os núcleos usam a mesma memória acabam tornando um gargalo para aplicação. Por esse motivo o uso de todos os núcleos no I7 faz com que ele perca em desempenho para o *Core2Duo*, enquanto o *cluster Raspbery Pi* vence ambos.

Quando comparado o tempo de execução total o *cluster Raspberry PI* é 10,6 vezes mais rápido que o computador *Core2Duo* e 2 vazes mais rápido que o i7. Quando ao consumo de energia, o *cluster* é duas vezes mais eficiente que ambos os computadores.

3.2.8 *Performance Analysis of a Low Cost Cluster with Parallel Applications and ARM Processors*

Lima, Moreno e Dias (2016) iniciam seu trabalho falando sobre como o poder de processamento aumentou muito, no entanto a quantidade de dados processados também aumentou. No aumento da capacidade de processamento a lei de Moore deixou de ser aplicável, devido as limitações físicas enfrentadas na construção de transistores cada vez menores. Por conta disso

deixou de ser objetivo principal dos *designers* de processadores o aumento de desempenho de um único processador e sim a exploração de soluções paralelas (ver Seção 2.1).

Dessa forma o trabalho de Lima, Moreno e Dias (2016) busca explorar o paralelismo na Computação de Alto Desempenho. Para isso, foi construído um *cluster* formado por quatro *Raspberry PI* onde a comunicação entre os nós é feita por uma rede local LAN (*Local Area Network*) e o aproveitamento do paralelismo é dado pelo uso de bibliotecas MPI. Durante o trabalho foi realizado experimentos de multiplicação de matrizes e de cálculo de produto escalar. Com a ajuda do *benchmark* HPL foi possível obter de desempenho como o tempo de execução e *Gflops*.

Quanto ao ambiente de teste, os testes foram realizados em um *cluster* formado por quatro *Raspberry PI 2 Model B*, todos usando como Sistema Operacional o *Raspbian*. Todos os nós foram conectados em um *switch* de rede local e toda a comunicação entre os nós se deu via protocolo SSH (*Secure Shell*). Para calcular o desempenho médio os autores usaram a função *MPI_Wtime()* disponível na API MPI para calcular o tempo médio de execução dos algoritmos depois de 100 execuções.

Com base em simulações com diferentes números de nós e ordem das matrizes, ficou constatado que o *cluster* tem um desempenho superior na execução das multiplicações de matrizes, chegando a 66% de melhora quando comparado o desempenho do *cluster* de quatro nós, em relação algoritmo sequencial em um nós. Foi constatado que quando as matrizes tem ordem de 100 não há ganho de desempenho quando o número de nós é incrementado. Isso é dado pelo fato que para quantidades pequenas de dados o tempo gasto com troca de informações entre os nós é maior que o tempo de processamento interno. Também é visto que o *speed-up* [2] aumenta junto com o número de nós para matrizes da mesma ordem.

Nas simulações de produto escalar de dois vetores com tamanho 100, 250 e 500 pode-ser concluir que a paralelização representou um ganho de 52%. Ao analisar o tempo de execução dos vetores de ordem cem pode-se concluir que não houve mudanças significativas de desempenho. No entanto ao realizar comparações com vetores maiores é possível perceber um aumento no desempenho do *cluster*.

Na execução do *benckmark* HPL foi realizado a resolução de um sistema linear denso de ordem 4000 para medir o tempo necessário para a resolução. Além disso é possível obter usando o HPL o número de operações de ponto flutuante por segundo (*Gflops*). Também nesse cenário de testes ficou comprovado que há um ganho de desempenho em relação a execução sequencial para a execução paralela com quatro nós. Neste caso, foi observado que a execução paralela foi 64% mais rápida. Adicionalmente foi possível concluir que a arquitetura ARM tem um bom desempenho na execução de operações matemáticas massivas. Com relação ao número de operações realizadas, ouve um aumento de aproximadamente 65%. Enquanto o número de

[2] *speed-up* em arquitetura de computadores é uma medida referente a performance de dois sistemas para o mesmo problema.

operações com um nó é um pouco menor de 300 *Gflops*, para quatro nós o número foi superior a 1000 *Gflops*

As conclusões dos autores foram que: O aumento no número de nós implica na redução considerável do tempo de execução; Um aumento na quantidade de dados reflete no ganho de desempenho do *cluster*, dado que para quantidade pequenas de dados o tempo gasto na comunicação é maior que o tempo gasto com o processamento; E finalmente, com base nas aplicações executadas juntamente com o teste de *benchmark* realizado conclui-se que o *cluster Raspberry PI* tem bom desempenho na execução de algoritmos paralelos.

Tabela 1 – Comparativo entre os estudos relacionados

Título	Ano	Numero de nós	Modelo do Raspberry PI	Tipo de Cluster	Benchmark
The Glasgow Raspberry Pi Cloud: A Scale Model for Cloud Computing Infrastructures	2013	56	*Raspberry PI 1 Model B*	Nuvem	Não Usou
A Container-based Edge Cloud PaaS Architecture based on RaspberryPi Clusters	2016	300	*Raspberry PI 2 Model B*	Nuvem	Não usou
Modelling Low Power Compute Clusters for Cloud Simulation	2017	12	Raspberry PI2 Model B	Nuvem	Não Usou
Comparison of Load Balancing Methods for Raspberry Pi Clustered Embedded Web Servers	2016	3	Raspberry PI2 Model B	Apache Web Server	Não Usou
Study of Raspberry Pi 2 Quad-core Cortex-A7 CPU Cluster as a Mini Supercomputer	2016	4	Raspberry PI 2 Model B	Alto Desempenho	*High Performance Linpack*
Performance of a Low Cost Hadoop Cluster for Image Analysis in Cloud Robotics Environment	2016	20	Raspberry PI 2 Model B	Alto Desempenho Usando Apache Hadoop	Definiu um benchmark próprio
Avaliação de Cluster Raspberry Pi para Execução de Aplicações de Análise de Imagens Microscópicas Médicas	2016	16	Raspberry PI 2 Model B	Alto Desempenho usando MPI	Definiu um benchmark próprio
Performance Analysis of a Low Cost Cluster with Parallel Applications and ARM Processors	2016	4	Raspberry PI2 Model B	Alto Desempenho Usando MPI	*High Performance Linpack*

3.3 Interpretação dos Resultados

Findada a análise dos artigos encontrados feita na Seção 3.2 podem ser feitas considerações acerca do uso de computadores *Raspberry PI* na construção de *clusters* e o seu uso na computação de Alto Desempenho. Foram encontradas três usos desses *clusters*, na construção de nuvens, a análise de um *Apache Web Server* e o uso na Computação de Alto Desempenho sobretudo fazendo uso das bibliotecas MPI e do *framework Apache Hadoop*.

A partir dos trabalhos de (TSO et al., 2013; PAHL et al., 2016; KECSKEMETI; HAJJI; TSO, 2017) que foram focados no uso de *Raspberry PI* para a construção de nuvens. Como a construção de uma nuvem não é um dos objetivos deste trabalho, os três artigos citados não apresentam contribuições significativas para a realização de Computação de Alto Desempenho, visto que o objetivo deste trabalho é analisar o desempenho para grandes quantidade de dados e principalmente a realização de cálculos. Como pontos positivos retirados da leitura deste trabalhos é possível indicar que além do baixo custo, *clusters Raspberry PI* são escaláveis e consumem uma pequena quantidade de energia, além da impossibilidade de usar aplicações de arquitetura x86 diretamente nos computadores ARM.

Mesmo o trabalho de (MADURANGA; RAGEL, 2016) sendo focado na construção de um *Web Server Apache*, já foi possível identificar um problema na construção de um *cluster* de Alto Desempenho. O problema é que quando a quantidade de usuários no servidor era pequena e não havia mudanças significativas no desempenho. Isso é dado pela demora de comunicação entre os nós ser maior que o tempo de processamento interno. Este problema foi confirmado com a leitura dos outros artigos selecionados.

Os trabalhos (MAPPUJI et al., 2016; QURESHI et al., 2016; RAMOS; RALHA; TEODORO, 2016; LIMA; MORENO; DIAS, 2016) foram dentre os encontrados os arquivos mais significativos para esse trabalho. Especialmente Mappuji et al. (2016) falam sobre o custo como um fator importante para o estudo de Computação de Alto Desempenho em computadores ARM, enquanto Lima, Moreno e Dias (2016) trazem o problema que motiva esse trabalho como uma das suas motivações, isso é, mesmo com a capacidade de processamento aumentando o aumento de dados para serem processados também aumentou muito.

O trabalho de Mappuji et al. (2016) ajudou a identificar que o aumento de *cores* em um único *Raspberry PI* não apresenta contribuições significativas no desempenho. Desta maneira conclui-se que a melhor forma de explorar o paralelismo está no número de nós e não na quantidade de *cores* no processador. Também foi identificado que o uso de *overclocking* no processador ARM representa ganhos de desempenho.

Os trabalhos de Qureshi et al. (2016) e Ramos, Ralha e Teodoro (2016) foram semelhantes ao fazer análise de imagens, porém usando métodos diferentes para aplicações diferentes. Enquanto o primeiro usou o *framework Apache Hadoop* o segundo uso as bibliotecas MPI. O primeiro trabalho confirmou o que já havia sido encontrado em (MAPPUJI et al., 2016) que o

overclocking representa um ganho de desempenho e que o desempenho de um *cluster Raspberry* é inferior ao uso de máquinas virtuais. Já o segundo identificou que o *cluster* construído é melhor do que o desempenho sequencial em computadores *Core2Duo* e I7. Ambos trabalhos confirmaram que a comunicação e o tempo de acesso a memória são gargalos.

A partir das conclusões de (LIMA; MORENO; DIAS, 2016) conclui-se que um *cluster* formado por Raspbery PI apresenta um bom desempenho nas operações matemáticas de multiplicação de matrizes e do produto vetorial. Também confirma que o ganho de desempenho está relacionado a quantidade de nós, aliado disso o desempenho é melhor para grandes quantidades de dados.

A partir da revisão sistemática concluem-se os seguintes pontos positivos na construção de um *cluster Raspberry PI*:

- Escalabilidade
- Baixo Custo
- Bom desempenho em operações matemáticas
- O *overclocking* do processador representa ganhos significativos.
- Baixo consumo de energia

E os seguintes pontos negativos:

- Operações em disco são lentas
- A comunicação entre os nós é um gargalo
- Falta de aplicações que estão disponíveis apenas para x86.

Com os pontos positivos e negativos encontrados nesta Revisão Sistemática é possível concluir que o sistema embarcado *Raspberry PI* apresenta-se como uma opção viável para a realização de uma Computação de Alto Desempenho. Os trabalhos mais relevantes encontrado na revisão foram os de Qureshi et al. (2016), Ramos, Ralha e Teodoro (2016) e Lima, Moreno e Dias (2016). Com a semelhança de tópicos ao comparar à análise de imagem, mesmo em aplicações diferentes apresentadas em Qureshi et al. (2016) e (RAMOS; RALHA; TEODORO, 2016), tem-se uma análise de performance entre o *framework Apache Hadoop* e o protocolo MPI. Uma análise deste tipo não é possivel em relação a multiplicação de matrizes apresentada em (LIMA; MORENO; DIAS, 2016). Por tanto esse trabalho trará a implementação da multiplicação de matrizes no *Apache Hadoop* e comparará com os resultados obtidos usando MPI.

4

Multiplicação de Matrizes

Este capítulo apresenta de forma detalhada os experimentos e elementos fundamentais para o estudo da multiplicação de matrizes conforme definido na Revisão Sistemática do Capítulo 3.

Há indícios que cálculos realizados com ajuda de tabelas já venham sendo utilizados desde da antiguidade, com as primeiras técnicas tendo sido empregadas por chineses em 2500 a.C. No entanto, foi o matemático inglês James Joseph Syvester, que cunhou o termo matriz para denominar esses cálculos. Mesmo tendo um conceito fundamentalmente matemático, o uso de operações sobre matrizes permeia outras áreas, dentre elas: as engenharias de modo geral, a economia e a computação (LEVORATO, 2017).

Na computação, o uso de matrizes é amplo, com aplicações simples para representar uma estrutura em um programa, onde uma tabela de disciplinas e notas seja armazenada em linhas e colunas de uma matriz. Na computação, a subárea que faz um uso em grande escala de matrizes é no processamento de imagens visto que uma imagem é representada como uma matriz de *pixels* (PEDRINI; SCHWARTZ, 2008). Com o uso em larga escala de matrizes nessas áreas, cabe a Computação de Alto Desempenho o estudo de formas mais rápidas de realizar as operações disponíveis nos conceitos matemáticos de matrizes de forma mais rápida possível. Desta maneira, este capítulo mostra duas alternativas de realizar a operação de multiplicação de matrizes. A primeira alternativa é um *cluster* de *Raspberry PI* baseado na plataforma *Apache Hadoop*. Pontos como complexidade do algoritmo, tempos de leitura e escrita e diferentes tamanhos de matrizes serão levados em consideração. A segunda alternativa usada para fins de comparação de tempo de execução será multiplicação paralela de matrizes usando a biblioteca *OpenMPI*.

Este capitulo está dividido seguindo a estrutura: A Seção 4.1 apresenta os conceitos matemáticos de uma matriz, suas características e operações. Na Seção 4.2 fundamenta porque a operação de multiplicação é uma operação naturalmente paralelizável e em seguida na Seção 4.3 é mostrado como matrizes podem ser multiplicadas usando o conceito de *MapReduce*. Na Seção

4.4, consideramos o desempenho de estudos relacionados na comparação de *Hadoop* com MPÌ.

4.1 Definição Matemática de uma Matriz

Dados dois números inteiros M e N, ambos maiores do que 1. Uma matriz é o nome dado a dupla sequência de números reais distribuídos em M linhas e N colunas (CALLIOLI; DOMINGUES; COSTA, 1987). Uma matriz é representada da seguinte maneira:

$$A_{mn} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{(m-1)1} & a_{(m-1)2} & \dots & a_{(m-1)n} \\ a_{m1} & a_{m2} & \dots & a_{m2} \end{bmatrix}$$

Onde uma linha M é:

$$M = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1(m-1)} & a_{1m} \end{bmatrix}$$

E uma coluna N:

$$N = \begin{bmatrix} a_{11} \\ a_{21} \\ \dots \\ a_{(m-1)1} \\ a_{m1} \end{bmatrix}$$

Os elementos de uma matriz são chamados de escalares e na maioria das vezes são números reais ou complexos. Geralmente um escalar é representado na forma a_{ij} onde i representa a linha e j a coluna. Costumeiramente uma matriz é representada por uma letra maiúscula (LEON, 1999)

4.1.1 Operações sobre matrizes

É possível realizar três operações sobre matrizes, são elas: adição, multiplicação por escalar e multiplicação de duas matrizes. Além dessas três operações também é possível verificar igualdade entre duas matrizes (LEON, 1999; CALLIOLI; DOMINGUES; COSTA, 1987). Além dessas três operações clássicas sobre as matrizes também existem outras operações, dentre elas estão; O calculo da transposta, inversa, diagonal principal, dentre outras.

4.1.1.1 Igualdade

Duas matrizes A e B de tamanho M x N são consideradas iguais se, somente se, para todo $a_{ij} = b_{ij}$

Ex 1:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

4.1.1.2 Adição

Dado duas matrizes A e B, a soma delas é uma terceira matriz C onde todo escalar c_{ij} = a_{ij}+b_{ij}

Ex 2:

$$A = \begin{bmatrix} 2 \\ 3 \\ 6 \\ 8 \end{bmatrix} B = \begin{bmatrix} 1 \\ 3 \\ 5 \\ 7 \end{bmatrix} C = \begin{bmatrix} 3 \\ 6 \\ 11 \\ 15 \end{bmatrix}$$

4.1.1.3 Multiplicação por número

Seja λ um número qualquer e A uma matriz. A multiplicação (A x λ) é dada por λA = λa_{ij} para todo a_{ij} pertencente a A.

Ex 3 :

$$2 * A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} 2A = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

4.1.1.4 Multiplicação entre duas matrizes

Dadas duas matrizes A e B de tamanhos M x N e M x P elas podem ser multiplicadas entre si, desde que os valores de M e P sejam iguais, o resultado da multiplicação é a matriz C de tamanho M x P.

Ex 4:

$$A = \begin{bmatrix} 2 & 4 & 6 \\ 1 & 3 & 7 \\ 9 & 5 & 2 \end{bmatrix} B = \begin{bmatrix} 2 \\ 5 \\ 0 \end{bmatrix}$$

$$C = \begin{bmatrix} 2*2+4*5+6*0 \\ 1*2+3*5+7*0 \\ 9*2+5*5+2*0 \end{bmatrix} = \begin{bmatrix} 24 \\ 17 \\ 43 \end{bmatrix}$$

4.1.2 Tipos de matrizes

Algumas matrizes recebem nomes especiais, entre elas, as principais são as seguintes.

4.1.2.1 Matriz Coluna

Tem apenas uma coluna.

Ex 5:

$$A = \begin{bmatrix} 2 \\ 5 \\ 0 \end{bmatrix}$$

4.1.2.2 Matriz linha

Tem apenas uma linha.

Ex 6:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}$$

4.1.2.3 Matriz Nula

É uma matriz onde todos os seus escalares são zero.

Ex 7:

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

4.1.2.4 Matriz Quadrada

São as matrizes onde o número de linhas e colunas são iguais.

Ex 8:

$$A = \begin{bmatrix} 2 & 4 & 6 \\ 1 & 3 & 7 \\ 9 & 5 & 2 \end{bmatrix}$$

4.1.2.5 Matriz Diagonal

São as matrizes onde todos os elementos fora da diagonal principal são nulos.

Ex 9:

$$A = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

4.1.2.6 Matriz Identidade

É uma variação da matriz diagonal onde todos os elementos da diagonal principal são 1.

Ex 10:

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

4.1.2.7 Matriz transposta

A matriz transposta A' é obtida trocando as linhas por colunas de uma matriz A.

Ex 11:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} A' = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

4.1.3 Matrizes densas e matrizes esparsas

As matrizes podem ser classificadas como densas, isto é matrizes onde todos ou quase a totalidade dos seus escalares possuem valores não nulos. Já as matrizes esparsas são aquelas em que apenas uma pequena porcentagem dos escalares são válidos.

Ex 12 - Matriz 5 x 5 densa:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 22 & 24 & 25 \end{bmatrix}$$

Ex 13 - Matriz 5 x 5 esparsa:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

4.2 Porque a multiplicação de matrizes é paralelizável

Como exposto na Seção 4.1.1.4, a realização da multiplicação de duas matrizes A e B gera uma matriz C. Mas ao analisar detalhadamente a matriz C originada em 4.1.1.4 é possível identificar que o primeiro elemento da matriz C ($c_{11} = 24$) é gerado pelo somatório de cada elemento da linha 1 (2,4,6) de A multiplicado pelo elementos da coluna 1 (2,5,0) de B. E o mesmo se repete para os outros elementos de C.

Tal resultado não está relacionado a uma propriedade especifica das matrizes em questão, mas relacionado a definição da multiplicação de matrizes. Onde simplificando a linguagem matemática usada em livros como Leon (1999) e Callioli, Domingues e Costa (1987) chegamos a seguinte regra *para multiplicar duas matrizes multiplicamos cada linha de A pelas colunas de B*.

Quando implementa-se a multiplicação de matrizes em computadores a forma mais intuitiva é multiplicar linha 1 pelas colunas de B, e ir salvando esses resultados de forma sequencial. Essa implementação trivial pode ser vista através do pseudocódigo 1.

Algorithm 1 Multiplicação Sequencial de Matrizes

```
matrizA[M][N]
matrizB[M][P]
matrizC[M][P]
para i ← 1 até n faça
    para j ← 1 até n faça
        matrizC[i][j] ← 0
        para k ← 1 até n faça
            matrizC[i][j] ← matrizC[i][j] + matrizA[i][k] * MatrizB[k][j]
        fim para
    fim para
fim para
```

Em uma análise do pseudocódigo apresentado é possível identificar três laços aninhados, cada um desses laços tem tamanho n. Em termos de complexidade de algoritmos teremos então um O(n^3) para todos os casos. Com essa implementação temos uma solução ótima, isso é; para qualquer matriz que seja usada como entrada desse algoritmo, obteremos a matriz resultante com a resposta correta.

No entanto a Computação de Alto Desempenho lida com matrizes com milhares de linhas e colunas, usando tal implementação o tempo de espera para matrizes grandes cresce de forma exponencial. Com isso algumas aplicações seriam inviáveis, pois o tempo gasto para realizar a multiplicação seria maior que o tempo disponível.

O que torna esse algoritmo lento não se dá apenas pelo fato de ser sequencial. Além disso os cálculos são feitos de maneira local. Isto quer dizer, que para o calculo da linha n da matriz resultante,não precisa-se conhecer o resultado da linha um. Os resultados são completamente independentes.

Se for considerado na multiplicação de matrizes o produto de uma linha por coluna seja uma subestrutura ótima, é possível enquadrar facilmente isso como um problema de programação dinâmica conforme definido em Cormen et al. (2009). Onde esse problema pode ser mapeado de uma forma que cada linha seja computada como uma estrutura ótima e a tabela de resultados seria a matriz resultante.

Não é difícil compreender que esse simples mapeamento do problema de multiplicação de matrizes em um modelo de programação dinâmica não origina de forma alguma um ganho de desempenho, já que de maneira não tão rebuscada o Algoritmo 1 faz isso. A grande contribuição desse mapeamento é mostrar como podemos fazer a transição desse algoritmo para um algoritmo paralelo.

Para obter paralelismo em um computador com apenas uma CPU (ver Seção 2.1.1), é preciso o uso *threads*. Explicando o que foi definido em Tanenbaum e Bos (2014) uma *thread* é uma parte de um processo que executa uma tarefa especifica. Dessa forma é possível modelar

um processo que multiplique matrizes de forma que exista uma *thread* que divida a matriz em partes e outras n *threads* que executa o calculo de cada uma dessas partes.

Mas além do uso do pseudoparelelismo para o ganho de desempenho na multiplicação de matrizes também é possível realizar a multiplicação de matrizes em forma paralela em um *cluster* de alto desempenho. A multiplicação de matrizes paralela é possível usando os seguintes pontos.

- O nó mestre do *cluster* tem integralmente ambas as matrizes
- Os nós escravos recebem recebem a matriz B inteira e algumas linhas da matriz A (A quantidade de linhas enviadas para cada escravo é definida pela quantidade de escravos disponíveis).
- Os escravos fazem a multiplicação das linhas enviadas e enviam ao mestre
- O mestre condensa as linhas recebidas em uma matriz C.

Os pseudocódigos 2 e 3 mostram as funções de um mestre e um escravo.

Algorithm 2 Função do Mestre na Multiplicação de Matrizes

```
matrizA[M][N]
matrizB[M][P]
matrizC[M][P]          ▷ Inicializa as matrizes A e B, enquanto a matriz C continua Vazia
QuantidadeDeEscravos   ▷ Recebe esse Valor como um parâmetro
LinhasPorEscravo ← M/QuantidadeDeEscravo
para i ← 1 até QuantidadeDeEscravos faça
        ▷ Envia a matriz B para o Escravo em questão e depois envia o numero de linhas da
matriz A definido pela variável Linhas Por Escravo
fim para
para i ← 1 até QuantidadeDeEscravos faça
                       ▷ Recebe as linhas Multiplicadas e monta a matriz C
fim para
```

Uma implementação destes peseudocódigos será apresentada no capítulo 5

4.3 A multiplicação de matrizes usando funções de *map* e *reduce*

O *MapReduce* é um modelo de programação desenvolvido por Dean e Ghemawat (2008) com o objetivo de facilitar a programação paralela nos servidores da Google. A ideia principal é lidar com grandes quantidades de dados distribuídos entre centenas ou milhares de máquinas de forma otimizada fazendo uso de codificações mais simples. A maior parte dessa simplificação se

Algorithm 3 Função do Escravo na Multiplicação de Matrizes

```
MatrizA[LinhasPorEscravo][N]
MatrizB[M][P]
Matrizc[M][P]  ▷ Recebe as Matrizes enviadas pelo mestre. Onde a matriz A é uma parte da matriz e a matriz B é a matriz inteira
para k ← 1 até M faça
    para i ← 1 até Quantidade de Linhas faça
        para j ← 1 até M faça
            MatrizC[i][k] ← MatrizC[i][k] + MatrizA[i][j] * MatrizB[j][k]
        fim para
    fim para
fim para
                    ▷ Após o calculo as linhas calculadas da matriz C são enviadas ao mestre
```

dá pelo uso de funções de *map* e *reduce* disponíveis em linguagens de programação sequencial como Lisp.

No modelo usado, os dados de entrada são tratados na função de *Map*, onde são tratados de uma maneira definida pelo usuário para que seja possível gerar um par de chaves. Esse par de chaves será formado por uma chave intermediária e um valor associado a esse par de chaves. Um exemplo de função *Map* é a função de mapeamento em um programa que conte o número de ocorrências de uma determinada palavra em um texto[1].

Algorithm 4 Função de Map em um WordCount

```
procedure MAP(key,value)                    ▷ Key = Nome do Documento
                                            ▷ Value= Conteudo do Documento
    para w ← i até Value faça
                                            ▷ Onde w é uma palavra no Documento
     GeraKey(w,"1")
    fim para
fim procedure
```

Nesta função cada palavra em um documento será associada a sua chave, e então as chaves serão divididas entre os computadores escravos disponíveis e a função de *Reduce* será aplicada. Neste caso, a função de *Reduce* é quem realiza o calculo, de forma que a função *Map* funciona como função de preparação para a função *Reduce*.

Dessa forma o *MapReduce* para a contagem de palavras irá receber o arquivo contendo o texto, em seguida irá dividir dividir o texto em partes menores e realizar a função de *Map* em cada uma dessas partes e por fim aplicar o *Reduce*. Um fluxograma deste processo pode ser visto na Figura 2

Muitas implementações do *MapReduce* existem. Neste trabalho será a utilizada a implementação disponibilizada pelo *Apache Hadoop* (ver Seção 2.6). Nesta implementação, o

[1] Os pseudocódigos das funções de *Map* e *Reduce* descritos nesta seção são baseados nos pseudocódigos disponibilizados por Dean e Ghemawat (2008)

Algorithm 5 Função de *Reduce* em um *WordCount*

procedure REDUCE(key,values)
▷ key = uma palavra
▷ Values= um iterador com os valores associados a essa palavra
Result ← 0
para i ← 1 **até** values **faça**
Result ← $Result + values[i]$
fim para
Escreve(Result)
fim procedure

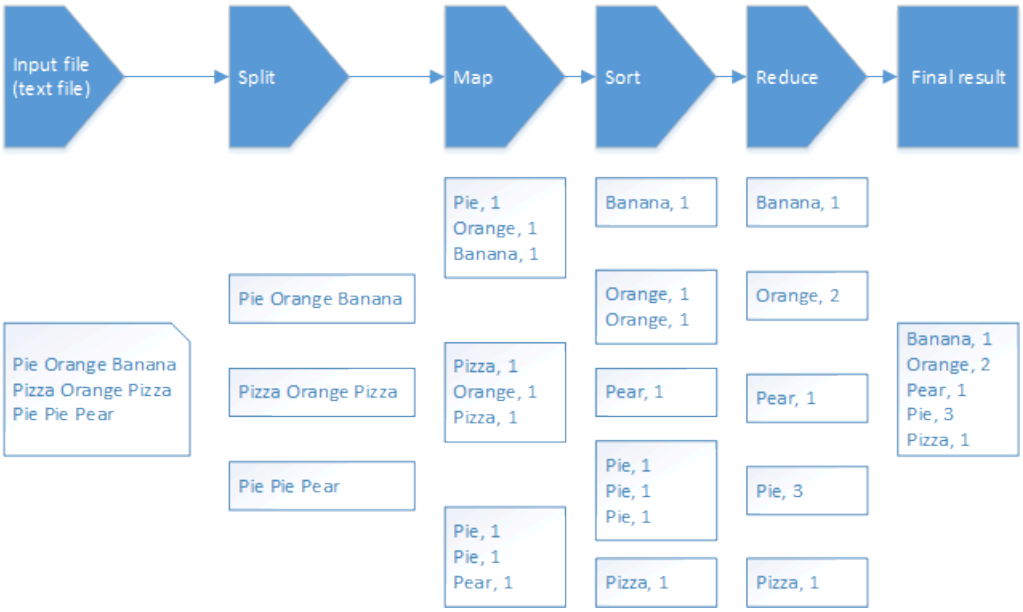

Figura 2 – Fluxograma do WordCount

Fonte – Widriksson (2014)

MapReduce trabalha em conjunto com o sistema de arquivos distribuído HDFS e o número de divisões realizada pela função de *map* é definido pelo tamanho do arquivo de entrada divido pelo tamanho de bloco do HDFS.

Os arquivos de entrada para execuções no *Hadoop* são geralmente arquivos de texto e os de saída idem, mas novos tipos de dados poder ser utilizados como já definido em (QURESHI et al., 2016). Dentre esses novos tipos estão arquivos em formato de imagens, áudio e vídeo.

Conforme já dito o objetivo de criação do modelo *MapReduce* é a facilidade de paralelizar tarefas e também já é conhecido pela Seção 4.2. Desta forma busca-se encontrar a implementação de uma multiplicação de matrizes usando as funções de *Map* e Reduce.

Uma solução para este problema foi proposta por Aytekin (2015), onde as matrizes que devem ser multiplicadas M e N são mapeadas como a uma tupla da forma M(I,J,V) e N(I,J,V) onde i e j representam respectivamente o numero da linha e da coluna. E a multiplicação dessas duas matrizes gerará uma matriz P também mapeada da forma P(I,J,V). O pseudocódigo da

Algorithm 1: The Map Function

1 **for** *each element* m_{ij} *of* M **do**
2 produce $(key, value)$ pairs as $((i,k),(M,j,m_{ij}))$, for $k = 1,2,3,..$ up to the number of columns of N
3 **for** *each element* n_{jk} *of* N **do**
4 produce $(key, value)$ pairs as $((i,k),(N,j,n_{jk}))$, for $i = 1,2,3,...$ up to the number of rows of M
5 **return** *Set of (key, value) pairs that each key,* (i,k)*, has a list with values* (M,j,m_{ij}) *and* (N,j,n_{jk}) *for all possible values of* j

Algorithm 2: The Reduce Function

1 **for** *each key (i,k)* **do**
2 sort values begin with M by j in $list_M$
3 sort values begin with N by j in $list_N$
4 multiply m_{ij} and n_{jk} for j_{th} value of each list
5 sum up $m_{ij} * n_{jk}$
6 **return** $(i,k), \sum_{j=1} m_{ij} * n_{jk}$

Figura 3 – Função *Map* e *Reduce* para a Multiplicação de Matrizes

Fonte – Disponível em Aytekin (2015)

multiplicação é apresentado na Figura 3.

A seguir os passos desta multiplicação usando matrizes M de tamanho 2x3 e N de tamanho 3x2 serão mostrados exatamente como definidos por Aytekin (2015):

Sejam as matrizes:

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} N = \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}$$

Logo a matriz gerada terá a forma:

$$M = \begin{bmatrix} 1a+2c+3e & 1b+2d+3f \\ 4a+5c+6e & 4b+5d+6f \end{bmatrix}$$

A função de mapeamento produzirá os seguintes pares de chaves para a matriz M:

$(i,k)(M,j,m_{ij})$

$M_{11} = 1$

$(1,1)(M,1,1)k = 1$

$(1,2)(M,1,1)k = 2$

$M_{12} = 2$

$(1,1)(M,2,2)k = 1$

$(1,2)(M,2,2)k = 2$

O processo se repetirá de forma idêntica para todos os elementos da matriz.

$M23 = 6$

$(2,1)(M,3,6)k = 1$

$(2,2)(M,3,6)k = 2$

Da mesma maneira a matriz N será mapeada:

$(i,k)(N,j,n_{jk})$

$N_{11} = a$

$(1,1)(N,1,a)i = 1$

$(2,1)(N,1,a)i = 2$

$N_{21} = c$

$(1,1)(N,2,c)i = 1$

$(2,1)(N,2,c)i = 2$

$N_{31} = e$

$(1,1)(N,3,e)i = 1$

$(2,1)(N,3,e)i = 2$

$N_{32} = f$

$(1,1)(N,3,f)i = 1$

$(2,1)(N,3,f)i = 2$

Por fim os pares de chaves e valores serão combinados da seguinte maneira:

$(i,k)[(M,j,m_{ij}),(M,j,m_{ij}),...(N,j,n_{jk}),(N,j,n_{jk})...]$

$(1,1)[(M,1,1),(M,2,2),(M,3,3),(N,1,a),(N,2,c),(N,3,e)]$

$(1,2)[(M,1,1),(M,2,2),(M,3,3),(N,1,b),(N,2,d),(N,3,f)]$

$(2,1)[(M,1,4),(M,2,5),(M,3,6),(N,1,a),(N,2,c),(N,3,e)]$

$(2,2)[(M,1,4),(M,2,5),(M,3,6),(N,1,b),(N,2,d),(N,3,f)]$

Então a função de *Reduce* será executada:

Primeiro cada conjunto de valores sera separado em duas listas do tipo:

$Lista_m = [(M,1,1),(M,2,2),(M,3,3)]$

$Lista_n = [(N,1,a),(N,2,c),(N,3,e)]$

e então os valores de P(i,k) são calculados:

$P(i,k) = \sum_{j=1} m_{ij} * n_{jk}$

Logo para todos valores de i e k, termos:

P(1,1) = 1a + 2c +3e

P(1,2) = 1b + 2d +3f

P(2,1) = 4a + 5c +6e

P(2,2) = 4b + 5d + 6f

No capítulo 5 será abordado a implementação deste algoritmo, bem como seus possíveis problemas.

4.4 Considerações sobre desempenho comparativo do *Apache Hadoop* em relação a MPI

Conforme definido na revisão sistemática realizada no capítulo 3, a parte prática do presente trabalho será uma comparação de desempenho entre um programa multiplicador de matrizes usando o *framework Apache Hadoop* e o protocolo *Messaging Passing interface(MPI)* usando a implementação disponibilizada pelo *OpenMPI* (ver Seção 2.7).

Duas métricas são as mais importantes quando falamos em Computação de Alto Desempenho, o tempo de execução e o consumo de energia. Por fins de simplificação apenas será abordado o tempo gasto na execução dos programas de multiplicação de matrizes.

Sabe-se que as características do *Hadoop* em associação com o modelo de programação do *MapReduce* fornece aos programadores as principais características presentes em um Sistema Distribuído (ver seção 2.3), principalmente heterogeneidade e resistência a falhas.

Um *Cluster* baseado em *Hadoop* terá portando uma robustez inerente na execução de suas tarefas, coisa que não será replicada na execução usando o *OpenMPI*. Por exemplo, a desconexão de um nó escravo arbitrário poderá gerar uma falha que inutilize todo o *cluster OpenMPI*.

Como o foco do trabalho é verificar o desempenho de execução nos dois cenários, falhas desse tipo não serão levadas em consideração. No entanto, como o foco é o desempenho, foram encontradas evidencias em trabalhos anteriores que indicam uma vantagem do protocolo MPI em relação ao *framework Hadoop*.

Os experimentos realizados em (LIANG et al., 2014) mostram a comparação do desem-

penho do *Hadoop* em relação ao *Spark* e o *DataMPI*. Como o *Spark* não faz parte do escopo do trabalho seus resultados não serão levados em consideração. Quando ao *DataMPI* é pode ser explicado como uma implementação do protocolo MPI capaz de realizar trabalhos semelhantes ao *Hadoop*.

Os resultados apresentados por Liang et al. (2014) foram obtidos em um *cluster* de oito nós compostos cada um por: um computador usando um processador *Intel Xeon E5620 (2.4 GHz)* , com 16 GB de RAM e 150 GB de espaço livre em disco. E os seguintes algoritmos foram utilizados: *Text Sort , WordSort, WordCount e Grep*.

Para o algoritmo de *Text Sort* para um texto de 8GB, o tempo de execução usando *DataMPI* foi de 69 segundos e o do *Hadoop* 117 segundos, representando assim, uma diferença percentual de 69,5 %. No caso do *WordCount*, *DataMPI* levou 130 segundos e o *Hadoop* 275 segundos, numa diferença de 111,5 %. Na execução de *WordSort,Text Sort e Grep* para arquivos de 128MB o *hadoop* foi inferior em 54%.

Dois Pontos precisam ser destacados acerca do trabalho de Liang et al. (2014). Os computadores utilizados são computadores com arquitetura x86, não *Raspberry PI ARM* como os utilizados neste trabalho. Desta forma espera-se que o desempenho do *cluster Hadoop* em computadores *Raspberry PI* seja menor. O outro é que o *DataMPI* é uma implementação diferente de MPI da qual será abordada neste trabalho.

No trabalho de (QURESHI et al., 2016), o *cluster Raspbery PI* utilizou o *Hadoop*, portanto deste trabalho foram retirados as configurações do *hadoop* utilizadas neste trabalho (a configuração detalhada está presente no apêndice A), além disso foi constatado que o *Hadoop* tem uma perca acentuada de desempenho em relação a computadores convencionais a medida que os dados aumentam.

A partir da analise do trabalho de (RAMOS; RALHA; TEODORO, 2016), foi detectado que o tempo de leitura e escrita do *Raspberry* representa um gargalo. O gargalo representado pela entrada e saída no *Raspberry* deverá ser acentuado no *cluster hadoop* já que o sistema de arquivos compartilhado deverá ser mais lento. Tal gargalo deve-se ao fato do armazenamento do *Raspberry PI* usar *sd cards*, os quais mesmo o mais novos têm um desempenho de entrada e saída menor que os HDs dos computadores x86.

Os resultados anteriores apresentados nesta seção mostram indícios de uma possível perca de desempenho do *Hadoop* em relação ao MPI na multiplicação de matrizes. Além dos fatores aqui apresentados as implementações dos algoritmos apresentadas no Capítulo 5 tendem acentuar a diferença de desempenho.

5

Implementação de MM no *Hadoop* e *OpenMPI*

Neste capítulo serão apresentadas as duas implementações de multiplicação de matrizes, conforme mencionado no Capítulo 4.

Para a implementação usando *Apache Hadoop* foi usado como base os códigos disponibilizados por Aytekin (2015). O código é uma implementação do pseudocódigo apresentado na Figura 3. As execuções preliminares desta implementação mostraram que a maior parte do tempo de execução está concentrado na função de *map* (ver Seção 4.3). Desta forma, além da execução do código original de Aytekin (2015), serão usadas variações com objetivo de melhorar o tempo de execução da função *map*.

A implementação em MPI usou como base o código disponibilizado por Barney (2005). No entanto, algumas alterações foram feitas e serão detalhadas na Seção 5.3. Além de uma implementação na linguagem de programação C foi desenvolvida uma implementação em Java. A decisão por duas implementações em linguagens diferentes ocorreu com o objetivo de medir o quanto a linguagem interfere no tempo de execução do mesmo algoritmo.

O capítulo está organizado da seguinte maneira: na Seção 5.1 são mostrados os equipamentos utilizados. Nas Seções 5.2 e 5.3, respectivamente, são descritas as implementações em *Apache Hadoop* e *OpenMPI*.

5.1 Hardware Utilizado

Para a execução dos testes foram usados quatro *Raspberry PI 2 Model B*. Optou-se por usar apenas um núcleo de cada *Raspberry* (exeto no mestre, onde dois núcleos são usados) na execução do MPI visto que o trabalho de Mappuji et al. (2016) mostrou que o uso de mais núcleos e *overclock* não representam mudanças significativas de desempenho.

Em cada placa foi utilizado um cartão de memória *SanDisk Ultra class 10* com uma

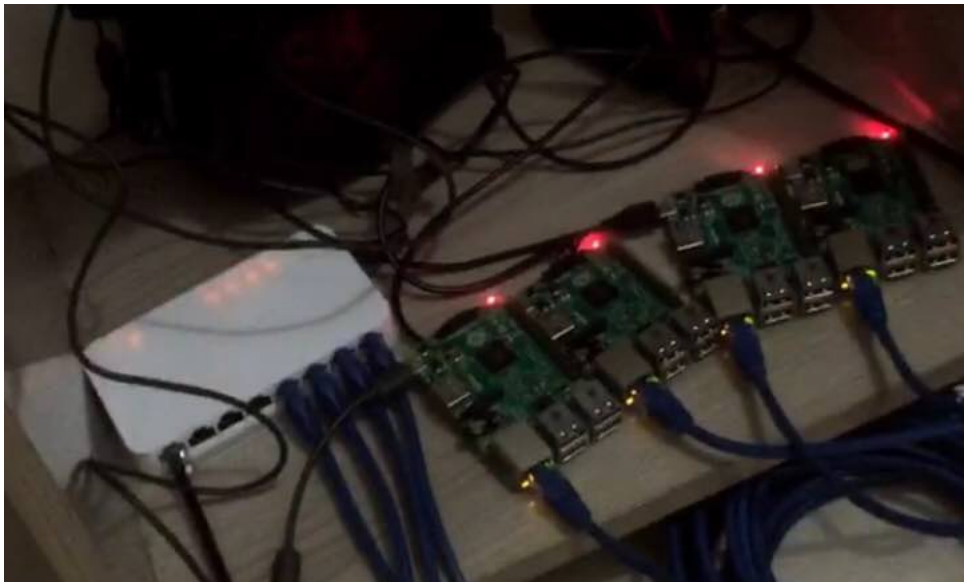

Figura 4 – *Hardware* do *cluster* utilizado no trabalho.

velocidade de leitura e escrita de 30MB por segundo, para minimizar o tempo de entrada e saída. Os computadores foram conectados em rede local *gigabit* usando o *switch D-Link DGS-1008A*.

5.2 Implementação em Java usando o *Apache Hadoop*

A fonte dos códigos tratados nesta seção foi disponibilizada por Aytekin (2015) [1], em um execução de teste dessa codificação foi possível verificar o acerto das respostas para a multiplicação de matrizes.

Também foi identificado que a maior parte do tempo de execução estava concentrado no tempo de execução da função *map*. Desta maneira esta função foi modificada pelo autor deste trabalho com intuito de melhorar o desempenho, as modificações realizadas são descritas na seguintes subseções. A função *Reduce* não foi modificada, sendo usada igualmente em todas as execuções.

5.2.1 A Função de Reduce

A função de *Reduce* no Código 1 é uma implementação do pseudocódigo da figura 3, ou seja, os pares de chaves são divididos em $lista_M$ e $lista_N$ e cada elemento da matriz resultante é calculado por um somatório.

Código 1 – Função *Reduce* disponibilizada por Aytekin (2015)

```
1 public void reduce(Text key, Iterable<Text> values, Context context)
2                         throws IOException, InterruptedException {
```

[1] Disponível em: https://github.com/marufaytekin/MatrixMultiply

```
3                 String[] value;
4                 //key=(i,k),
5                 //Values = [(M/N,j,V/W),..]
6                 HashMap<Integer, Integer> hashA = new HashMap<Integer,Integer>();
7                 HashMap<Integer,Integer> hashB = new HashMap<Integer, Integer>();
8                 for (Text val : values) {
9                         value = val.toString().split(",");
10                        if (value[0].equals("M")) {
11                                hashA.put(Integer.parseInt(value[1]),
                                   ↪  Integer.parseInt(value[2]));
12                        } else {
13                                hashB.put(Integer.parseInt(value[1]),
                                   ↪  Integer.parseInt(value[2]));
14                        }
15                }
16                int n = Integer.parseInt(context.getConfiguration().get("n"));
17                int result = 0;
18                int m_ij;
19                int n_jk;
20                for (int j = 0; j < n; j++) {
21                        m_ij = hashA.get(j);
22                        n_jk = hashB.get(j);
23                        result += m_ij * n_jk;
24                }
25                if (result != 0) {
26                        context.write(null,
27                                        new Text(key.toString() + "," + result));
28                }
29        }
30}
```

5.2.2 A Função *Map* Original

Na função original de Aytekin (2015) os elementos das matrizes são armazenados em dois arquivos M e N representando respectivamente as matrizes M e N. Cada elemento é armazenado como uma tupla da forma (M,i,j,e), onde 'M' representa a matriz, 'i' representa a linha, 'j' a coluna e 'e' representa o valor do elemento. Cada tupla é armazenada em uma linha do arquivo.

Código 2 – Função *Map* disponibilizada por Aytekin (2015)

```
1     public void map(LongWritable key, Text value, Context context)
2                         throws IOException, InterruptedException {
3                 Configuration conf = context.getConfiguration();
4                 int m = Integer.parseInt(conf.get("m"));
5                 int p = Integer.parseInt(conf.get("p"));
6                 String line = value.toString();
```

```
7                String[] indicesAndValue = line.split(",");
8                Text outputKey = new Text();
9                Text outputValue = new Text();
10               if (indicesAndValue[0].equals("M")) {
11                       for (int k = 0; k < p; k++) {
12                               outputKey.set(indicesAndValue[1] + "," + k);
13                               outputValue.set(indicesAndValue[0] + "," +
                                  ↪  indicesAndValue[2]
14                                                  + "," + indicesAndValue[3]);
15                               context.write(outputKey, outputValue);
16                       }
17               } else {
18                       for (int i = 0; i < m; i++) {
19                               outputKey.set(i + "," + indicesAndValue[2]);
20                               outputValue.set("N," + indicesAndValue[1] + ","
21                                                  + indicesAndValue[3]);
22                               context.write(outputKey, outputValue);
23                       }
24               }
25       }
```

5.2.3 A Função *Map* para matrizes em único arquivo

A primeira alternativa na tentativa de melhoria na função *map* é salvar os dois arquivos de entrada contendo uma matriz em cada em apenas um com as duas matrizes, e com isso não ter um tempo tão grande de espera para entrada e saída. A codificação para esse teste é exatamente a mesma em utilizada no código 2.

Essa abordagem pode trazer ganho em desempenho quando as matrizes são menores, pois como mostrado por Ramos, Ralha e Teodoro (2016) e Qureshi et al. (2016) a abertura e leitura de arquivos são gargalos, e dessa forma o arquivo é lido de maneira a não influenciar no desempenho, mas espera-se que quando os arquivos sejam maiores os tempos se equiparem visto que o *hadoop* faz um *split* dos arquivos maiores e executam em tarefas menores.

5.2.4 A função *Map* para matrizes salvas de maneira convencional

Quando matrizes são salvas em um arquivo a maneira mais comum de fazer isso é fazendo com uma linha contenha uma quantidade de elementos separados por espaços e esses elementos separados por espaço representam as colunas e cada linha do arquivo é uma linha da matriz. Na implementação base de Aytekin (2015) os arquivos não seguem este padrão, as matriz estão salvas como tuplas.

O segundo tópico de estudo foi então salvar as matrizes da forma mais convencionais e realizar o mapeamento para tuplas dentro do código. Esta implementação diminui de forma

considerável o tamanho do arquivo em disco, mas em contrapartida é necessário um laço a mais no código para realizar a transformação de um elemento qualquer para a tupla. A medida que as matrizes crescem, esse laço também cresce e é mais lento. Assim o desempenho deverá ser maior apenas para matrizes pequenas.

Para repetir o que foi feito nos arquivos com a transição de tupla já feita, foram realizados testes com as matrizes salvas e um e em dois arquivos. Os Códigos 3 e 4 são mostrados a seguir, assim sendo possível compreender as mudanças realizadas.

Código 3 – Função *Map* lendo dois arquivos não mapeados

```
1   public void map(LongWritable key, Text value, Context context)
2                   throws IOException, InterruptedException {
3           Configuration conf = context.getConfiguration();
4           int m = Integer.parseInt(conf.get("m"));
5           int p = Integer.parseInt(conf.get("p"));
6           String line = value.toString();
7           String fileName = ((FileSplit)
            ↪  context.getInputSplit()).getPath().getName();
8           String[] indicesAndValue = line.split(" ");
9           Text outputKey = new Text();
10          Text outputValue = new Text();
11          if (fileName.equals("M")) {
12                  for(int v=0 ; v<p;v++){
13                  for (int k = 0; k < p; k++) {
14                          outputKey.set(rowCount + "," + k);
15                          outputValue.set("M" + "," + v
16                                          + "," + indicesAndValue[v]);
17                          context.write(outputKey, outputValue);
18                  }}
19                  rowCount++;
20          } else {
21                  for(int v=0; v<m;v++){
22                  for (int i = 0; i < m; i++) {
23                          outputKey.set(i + "," + v);
24                          outputValue.set("N," + rowCountN+ ","
25                                          + indicesAndValue[v]);
26                          context.write(outputKey, outputValue);
27                  }
28          }
29          rowCountN++;
30          }
31      }
```

Código 4 – Função *Map* lendo um arquivo não mapeado

```
1           public void map(LongWritable key, Text value, Context context)
2                   throws IOException, InterruptedException {
3               Configuration conf = context.getConfiguration();
4               int m = Integer.parseInt(conf.get("m"));
5               int p = Integer.parseInt(conf.get("p"));
6               String line = value.toString();
7               String[] indicesAndValue = line.split(" ");
8               Text outputKey = new Text();
9               Text outputValue = new Text();
10              if (matrix==true) {
11                      for(int v=0 ; v<p;v++){
12                      for (int k = 0; k < p; k++) {
13                              outputKey.set(rowCount + "," + k);
14                              outputValue.set("M" + "," + v
15                                              + "," + indicesAndValue[v]);
16                              context.write(outputKey, outputValue);
17                      }}
18                      rowCount++;
19                      if (rowCount==p){
20                              rowCount=0;
21                              matrix=false;
22                      }
23              } else {
24                      for(int v=0; v<m;v++){
25                      for (int i = 0; i < m; i++) {
26                              outputKey.set(i + "," + v);
27                              outputValue.set("N," + rowCount+ ","
28                                              + indicesAndValue[v]);
29                              context.write(outputKey, outputValue);
30                      }
31              }
32              rowCount++;
33              }
34      }
```

5.2.5 A função *Map* salva em um arquivo

Como ultimo teste foi realizado o mapeamento para os pares de chave/valor necessários para a aplicação da função *Reduce* foi salvo em um arquivo.

Desta maneira, as matrizes já mapeadas em pares de valores foram salvas em dois arquivos e a função *Map* não faz nada além de enviar linha por linha do arquivo para a função *Reduce*. Assim sendo possível observar o tempo gasto pela função *Reduce*. O Código 5 é a explicação de como isto foi feito.

Código 5 – Função *Map* apenas acessando os arquivos

```
1     public void map(LongWritable key, Text value, Context context)
2                         throws IOException, InterruptedException {
3                 Configuration conf = context.getConfiguration();
4                 int m = Integer.parseInt(conf.get("m"));
5                 int p = Integer.parseInt(conf.get("p"));
6                 String line = value.toString()
7                 String[] indicesAndValue = line.split(" ");
8                 Text outputKey = new Text();
9                 Text outputValue = new Text();
10                outputKey.set(indicesAndValue[0]);
11                outputValue.set(indicesAndValue[1]);
12                context.write(outputKey,outputValue);
13        }
```

5.3 Implementação usando *OpenMPI*

A implementação do algoritmo multiplicador de matrizes de Barney (2005) é uma implementação dos Pseudocódigos 2 e 3. No entanto, nesse trabalho o código foi modificado para que as matrizes multiplicadas estejam salvas em arquivos.

O código também foi modificado de maneira que os tamanhos das matrizes de entrada, bem como os nomes dos arquivos que contém as matrizes sejam passados como parâmetros. A razão da modificação para leitura de matrizes a partir de arquivos se dá pela necessidade de comparação usando as mesmas matrizes. Além disso, a suspeita levantada por Qureshi et al. (2016) e Ramos, Ralha e Teodoro (2016) de que o tempo de entrada e saída seja um gargalo para o *Raspberry PI* levou a essa abordagem.

No entanto é preciso observar que na implementação usando o *OpenMPI* optou-se por não usar um sistema de arquivo distribuído já que os arquivos das matrizes são pequenos e torna possível armazenar apenas no mestre (Ver Pseudocódigos 2 e 3) e enviar via rede as partes dos escravos. Como o sistema de arquivos HDFS do *Apache Hadoop* é distribuído entre os nós isso causa uma lentidão inerente nas operações de entrada e saída.

A implementação de Barney (2005) foi realizada na linguagem de Programação C. É conhecido que C é uma linguagem de mais baixo nível e com isso tem um desempenho maior em operações aritméticas e em laços (PRECHELT, 2000). Em contrapartida, o *Apache Hadoop* foi implementado em Java, uma linguagem com nível mais alto e interpretada, por isso, mais lenta do que C (PRECHELT, 2000).

Para analisar a influência de desempenho ocasionado pela diferença de linguagens de programação, e aproveitando a opção do *OpenMPI* que disponibiliza uma implementação de MPI para java, desta forma foi possível realizar uma implementaação em Java. O código de Barney (2005) foi reescrito em Java.

5.3.1 Implementação na Linguagem C

A implementação em C baseada em Barney (2005) realizada neste trabalho é bastante simples. Consiste em uma decisão *se - senão*. Onde o *Se* realiza o trabalho do Mestre e o *senão* o dos escravos.

O programa recebe cinco parâmetros, os dois primeiros representam a localização dos arquivos das matrizes M e N, o terceiro é definido onde a matriz resultante será salva, no quarto é informado a ordem da matriz. O outro parâmetro usado pelo programa é a quantidade de *tasks* cujo o valor é definido de acordo com a chamada do *mpirun* (Ver apêndice B).

A *task* zero é atribuída como o Mestre do programa, ela abre os arquivos contendo as duas matrizes e apenas ela tem conhecimento da matriz A inteira,ou seja, a primeira matrizes usada na multiplicação, após isso o numero de linhas da matriz A é divida pela quantidade de *tasks* escravas restantes. A resultado da divisão é a quantidade de linhas da matriz A que será enviada as *tasks* escravas, juntamente das linhas de A a matriz B,que é a segunda matriz a ser multiplicada, é enviada com todas suas linhas e colunas.

O trabalho das *tasks* escravas consiste em usar a linhas da matriz recebidas do Mestre para multiplicar pelas colunas da matriz B e é seguida enviar as novas linhas calculadas para o Mestre. Após receber estas linhas o mestre então começa a montar a matriz resultante C. Quando a matriz está montada o resultado é escrito no arquivo de saída.

Código 6 – Código fonte do algoritmo de multiplicação de matrizes usando MPI de Barney (2005) com alterações do autor

```
1  #include "mpi.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  #define MASTER 0               /* taskid of first task */
6  #define FROM_MASTER 1          /* setting a message type */
7  #define FROM_WORKER 2          /* setting a message type */
8
9  int main (int argc, char** argv){
10
11 double start = MPI_Wtime();
12 int numtasks,              /* number of tasks in partition *
13     taskid,                /* a task identifier */
14     numworkers,            /* number of worker tasks */
15     source,                /* task id of message source */
16     dest,                  /* task id of message destination */
17     mtype,                 /* message type */
18     rows,                  /* rows of matrix A sent to each worker */
19     averow, extra, offset, /* used to determine rows sent to each worker */
20     i, j, k, rc;           /* misc */
```

```
21 int MATSIZE = atoi(argv[4]);
22 int a[MATSIZE][MATSIZE],           /* matrix A to be multiplied */
23     b[MATSIZE][MATSIZE],           /* matrix B to be multiplied */
24     c[MATSIZE][MATSIZE];           /* result matrix C */
25
26
27 MPI_Status status;
28 MPI_Init(&argc,&argv);
29 MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
30 MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
31
32 if (numtasks < 2 ) {
33   MPI_Abort(MPI_COMM_WORLD, rc);
34   exit(1);
35   }
36
37 numworkers = numtasks-1;
38
39 /************************** master task ************************************/
40    if (taskid == MASTER){
41       FILE* M = fopen(argv[1],"r");
42       FILE* N = fopen(argv[2],"r");
43       int s;
44       for (i=0; i<MATSIZE; i++){
45          for (j=0; j<MATSIZE; j++){
46             fscanf(M,"%d",&s);
47             a[i][j]= s;
48             fscanf(N,"%d",&s);
49             b[i][j]= s;
50          }
51          }
52
53       averow = MATSIZE/numworkers;
54       extra = MATSIZE%numworkers;
55       offset = 0;
56       mtype = FROM_MASTER;
57
58       for (dest=1; dest<=numworkers; dest++){
59          rows = (dest <= extra) ? averow+1 : averow;
60          MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
61          MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
62          MPI_Send(&a[offset][0], rows*MATSIZE, MPI_INT, dest, mtype,
63                   MPI_COMM_WORLD);
64          MPI_Send(&b, MATSIZE*MATSIZE, MPI_INT, dest, mtype, MPI_COMM_WORLD);
65          offset = offset + rows;
66       }
67
68       /* Receive results from worker tasks */
```

```
69        mtype = FROM_WORKER;
70        for (i=1; i<=numworkers; i++)
71        {
72           source = i;
73           MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
74           MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
75           MPI_Recv(&c[offset][0], rows*MATSIZE, MPI_INT, source, mtype,
76                    MPI_COMM_WORLD, &status);
77        }
78
79        double finish = MPI_Wtime();
80        printf("Done in %f seconds.\n", finish - start);
81     }
82
83     if (taskid > MASTER)
84     {
85        mtype = FROM_MASTER;
86        MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
87        MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
88        MPI_Recv(&a, rows*MATSIZE, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
89        MPI_Recv(&b, MATSIZE*MATSIZE, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
90
91        for (k=0; k<MATSIZE; k++)
92           for (i=0; i<rows; i++)
93           {
94              c[i][k] = 0.0;
95              for (j=0; j<MATSIZE; j++)
96                 c[i][k] = c[i][k] + a[i][j] * b[j][k];
97           }
98        mtype = FROM_WORKER;
99        MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
100       MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
101       MPI_Send(&c, rows*MATSIZE, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
102    }
103    MPI_Finalize();
104 }
```

5.3.2 Implementação na linguagem Java

A implementação na linguagem segue a mesma lógica da implementação na linguagem C, a maior mudança realizada deve-se a incapacidade da implementação de MPI para java de enviar vetores multidimensionais em uma única mensagem (SQUYRES, 2014).

A forma encontrada para contornar este problema foi mapear a matriz B como um vetor unidimensional do tamanho $matsize * matsize$ representando assim a quantidade de elementos da matriz e o acesso se deu pela formula $(linha * matsize + coluna)$. Já a matriz A continuou do

mesmo modo que na implementação em C, no entanto, as linhas tiveram que ser enviadas em mensagens separadas para os escravos. A mesma estrategia foi empregada no envio das linhas da matriz resultante C.

A quantidade maior de mensagens trocadas por esta implementação será um fator preponderante na diferença de desempenho entre as duas implementações.

Código 7 – Código para multiplicação usando MPI em Java

```
1  import mpi.*;
2  import java.io.FileInputStream;
3  import java.io.FileNotFoundException;
4  import java.io.FileWriter;
5  import java.io.IOException;
6  import java.io.PrintWriter;;
7  import java.io.FileReader;
8  import java.io.InputStream;
9  import java.io.BufferedReader;
10 class MPIMatrixMultiply{
11     private static int MASTER =0;
12     private static int FROM_MASTER =1;
13     private static int FROM_WORKER =2;
14    static public  void main(String[] args) throws MPIException,IOException{
15         MPI.Init(args);
16         int MATSIZE = Integer.parseInt(args[3]);
17         int numtasks;
18         int taskid;
19         int numworkers;
20         int source;
21         int dest;
22         int mtype;
23         int tag=50;
24         int rows[] = new int[1];
25         int averow;
26         int extra;
27         int offset[] = new int[1];
28         int i,j,k;
29         int a[][] = new int[MATSIZE][MATSIZE];
30         int b[] = new int[MATSIZE*MATSIZE];
31         int c[][] = new int[MATSIZE][MATSIZE];
32         Status status;
33
34         taskid = MPI.COMM_WORLD.getRank();
35         numtasks = MPI.COMM_WORLD.getSize();
36         if(numtasks <2){
37             System.out.println("Need at least two MPI tasks. Quitting..");
38             MPI.COMM_WORLD.abort(0);
39         }
```

```
40
41          numworkers = numtasks-1;
42
43
44          if(taskid == MASTER){
45              double start = MPI.wtime();
46              FileReader  f = new FileReader(args[0]);
47              FileReader  f2 = new FileReader(args[1]);
48              BufferedReader M = new BufferedReader(f);
49              BufferedReader N = new BufferedReader(f2);
50              try{
51                  for(i=0;i<MATSIZE;i++){
52                      String lineA; lineA = M.readLine();
53                      String lineB; lineB = N.readLine();
54                      String asRow[] = lineA.split(" ");
55                      String bsRow[] = lineB.split(" ");
56                      for(j=0;j<MATSIZE;j++){
57                          a[i][j] = Integer.parseInt(asRow[j]);
58                          b[(i*MATSIZE)+j] = Integer.parseInt(bsRow[j]);
59                      }
60                  }
61                  f.close();
62                  f2.close();
63              }catch(Exception ex){
64                  System.out.println("Reading file error!!");
65              }
66
67          averow = MATSIZE/numworkers;
68          extra = MATSIZE%numworkers;
69          offset[0] = 0;
70          mtype = FROM_MASTER;
71          for (dest=1; dest<=numworkers; dest++){
72              rows[0] = (dest <= extra) ? averow+1 : averow;
73              MPI.COMM_WORLD.send(offset,1,MPI.INT,dest,mtype);
74              MPI.COMM_WORLD.send(rows,1,MPI.INT,dest,mtype);
75              int A[] = new int[MATSIZE];
76              for (intz=0;z<rows[0];z++){
77              MPI.COMM_WORLD.send(a[offset[0]+z],MATSIZE,MPI.INT,dest,mtype);
78              }
79              MPI.COMM_WORLD.send(b,MATSIZE*MATSIZE,MPI.INT,dest,mtype);
80              offset[0] = offset[0] + rows[0];
81          }
82          mtype= FROM_WORKER;
83          for(i=1;i<=numworkers;i++){
84              source = i;
85              MPI.COMM_WORLD.recv(offset,1,MPI.INT,source,mtype);
86              MPI.COMM_WORLD.recv(rows,1,MPI.INT,source,mtype);
87              for(int z=0;z<rows[0];z++){
```

```
88              MPI.COMM_WORLD.recv(c[z],MATSIZE,MPI.INT,source,mtype);}
89          }
90
91          FileWriter fw = new FileWriter(args[2]);
92          PrintWriter p = new PrintWriter(fw);
93          for(i=0;i<MATSIZE;i++){
94                  p.printf("%n");
95                  for(j=0;j<MATSIZE;j++){
96                      p.printf("%d ",(int)c[i][j]);
97                  }
98              }
99          p.close();
100
101         double finish = MPI.wtime() - start;
102         System.out.println("Done in: " +finish + " seconds");
103     }
104         if(taskid > MASTER){
105             mtype = FROM_MASTER;
106             MPI.COMM_WORLD.recv(offset,1,MPI.INT,MASTER,mtype);
107             MPI.COMM_WORLD.recv(rows,1,MPI.INT,MASTER,mtype);
108             int A[][] = new int[MATSIZE][MATSIZE];
109             for(int z=0;z<rows[0];z++){
110             MPI.COMM_WORLD.recv(A[z],MATSIZE,MPI.INT,MASTER,mtype);
111             }
112             MPI.COMM_WORLD.recv(b,MATSIZE*MATSIZE,MPI.INT,MASTER,mtype);
113
114             for (k=0; k<MATSIZE; k++)
115                 for (i=0; i<rows[0]; i++){
116                     c[i][k] = 0;
117                         for (j=0; j<MATSIZE; j++){
118                             c[i][k] = c[i][k] + A[i][j] * b[(j*MATSIZE)+k];}}
119                 }
120             mtype = FROM_WORKER;
121             MPI.COMM_WORLD.send(offset,1,MPI.INT,MASTER,mtype);
122             MPI.COMM_WORLD.send(rows,1,MPI.INT,MASTER,mtype);
123             for(int z=0;z<rows[0];z++){
124             MPI.COMM_WORLD.send(c[z],MATSIZE,MPI.INT,MASTER,mtype);
125             }
126         }
127         MPI.Finalize();
128     }
129 }
```

6
Resultados

Neste Capítulo, apresentamos os resultados encontrados nas aplicações descritas anteriormente nos Capítulos 4 e 5 ao serem executadas em um *cluster* de baixo custo utilizando processadores ARM e a plataforma *Raspberry PI*. Para isso foram realizados teste utilizando o *framework Apache Hadoop*. Também foram realizados testes usando MPI com a implementação do *OpenMPI* com implementações de um mesmo algoritmo em linguagens distintas.

6.1 Metodologia dos Experimentos

O trabalho faz a comparação de duas implementações distintas para multiplicação de matrizes, a primeira usa o *framework Apache Hadoop* e a segunda usa o *OpenMPI*. A descrição teórica de ambas implementações podem serem encontradas no Capítulo 4, enquanto o código fonte é descrito no Capítulo 5. As ordens de tamanho das escolhidas foram 50, 100, 150, 200 e 250, para os testes no *Hadoop* e 100, 200, 300, 400 e 500, para os testes com *OpenMPI*.

Foram geradas duas matrizes para cada uma das ordens escolhidas, onde optou-se por usar apenas números inteiros com o objetivo de evitar que operações de ponto flutuante, realizadas de maneiras distintas entre C e Java, fossem um fator preponderante das execuções. Cada uma das matrizes geradas foram salvas em arquivos cujo as colunas são separadas por um espaço. Após isso, um segundo *script* em Python mapeou como definido na seção 5.2 as matrizes para que pudessem ser usada na implementação do *Apache Hadoop*.

Conforme definido na Seção 4.1.3, as matrizes podem ser densas ou esparsas. Sabe-se que a grande maioria das matrizes usadas na computação são matrizes esparsas. Assim, para cada ordem de matrizes utilizadas nos testes foram definidas matrizes densas e esparsas. O fator de densidade escolhido para as matrizes esparsas foi de um por cento.

Como a implementação usando *OpenMPI* exige que a matriz seja salva de maneira completa, a multiplicação de matrizes esparsas gerará uma quantidade grande de multiplicações

e adições resultando em zero. O mesmo não ocorrerá na implementação usando o *Apache Hadoop* e, por isso, espera-se que para esses casos de teste a diferença entre as duas implementações seja menor.

O experimentos foram executados com a ajuda de um *script* escrito em Bash que realiza a chamada sucessiva de cada um dos algoritmos. Para cada ordem de tamanho das matrizes, tanto na execução no *Apache Hadoop* quanto no *OpenMPI*, foram realizadas um total de 12 chamadas, para ser possível obter uma média. Com os doze resultados foram retirados os *outliers*, isso é, o maior e menor valor. E então foi calculado a média aritmética do tempo de execução.

6.2 Resultados da execução no *Apache Hadoop*

A execução dos experimentos usando o *framework Apache Hadoop* foi dividida em duas partes. Na primeira e na segunda parte o tamanho do bloco do sistema de arquivos HDFS (Ver Seção 2.6) foi configurado com o tamanho padrão de 128MB.

Na primeira fase de execução foram usadas cada uma das mudanças na função de *Reduce* de Aytekin (2015) conforme descritas na Seção 5.2. Nas execuções desta fase foram usadas apenas matrizes densas. As execuções são rotuladas como A, B, C, D e E que, respectivamente, significam:

- A: As matrizes estão salvas em dois arquivos e cada linha representa uma tupla (chave,valor). Esta execução faz o uso do código original de Aytekin (2015) descrita no Código 2.
- B: As matrizes foram salvas em dois arquivos onde cada linha é uma linha da matriz a ser multiplicada. Essa execução faz o uso da função *Map* modificada pelo autor descrita no Código 3.
- C: As matrizes são representadas da mesma maneira que no Item B, no entanto, as duas matrizes estão salvas em apenas um arquivo. Essa execução utiliza o Código 4.
- D: A matrizes foram representadas da mesma maneira do Item A, no entanto, as duas matrizes estão salvas em apenas um arquivo. Essa execução utiliza o Código 2.
- E: As matrizes são salvas no arquivo com a função *Map* já realizada conforme descrito no código 5.

A segunda fase foi focada na execução de matrizes esparsas. Para essa fase foram utilizados apenas os experimentos A, D e E, pois o objetivo é verificar o ganho de desempenho no mapeamento da tupla (chave, valor) para a execução de matrizes esparsas. Dessa forma, foram salvos nos arquivos apenas aquelas posições da matrizes que contém números diferentes de 0. A densidade escolhida para cada uma das matrizes foi de 10%.

6.2.1 Resultados da primeira fase de testes

Na primeira fase de execuções usando o *Hadoop* o objetivo do autor é analisar de que forma o modelo de dados salvo em um arquivo de texto influencia no desempenho da execução da função de *map* no problema de multiplicação de matrizes, visto que nesta fase foi identificada em testes preliminares como responsável pela maior parte do tempo de execução do programa.

Um possível problema identificado no código de Aytekin (2015) foi o tamanho do arquivo de texto onde as matrizes estavam salvas. Portanto, as mudanças realizadas tiveram como foco alterar o tamanho dos arquivos e ler as matrizes de diferentes formas (As mudanças implementadas foram descritas na Seção 5.2). Com isso, foram utilizados o cinco testes descritos anteriormente nesse capítulo.

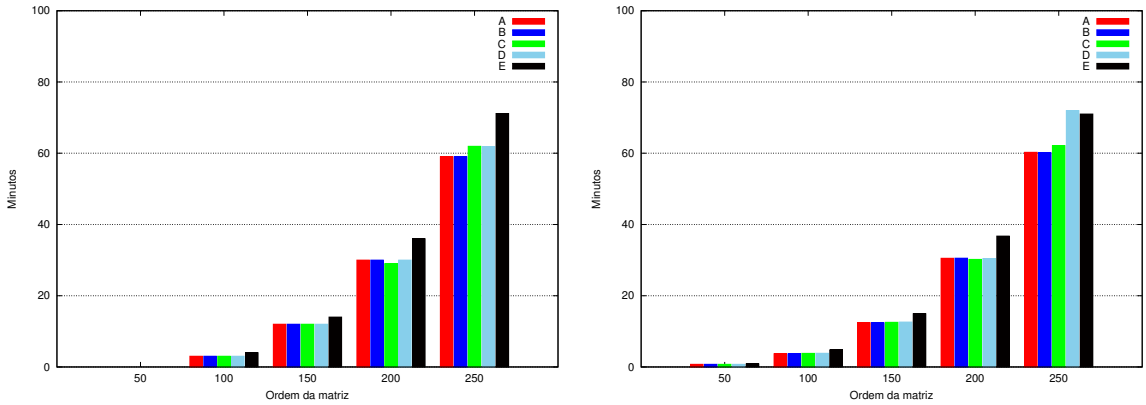

Figura 5 – Execução das matrizes densas no *Apache Hadoop* com um nó e dois nós, respectivamente.

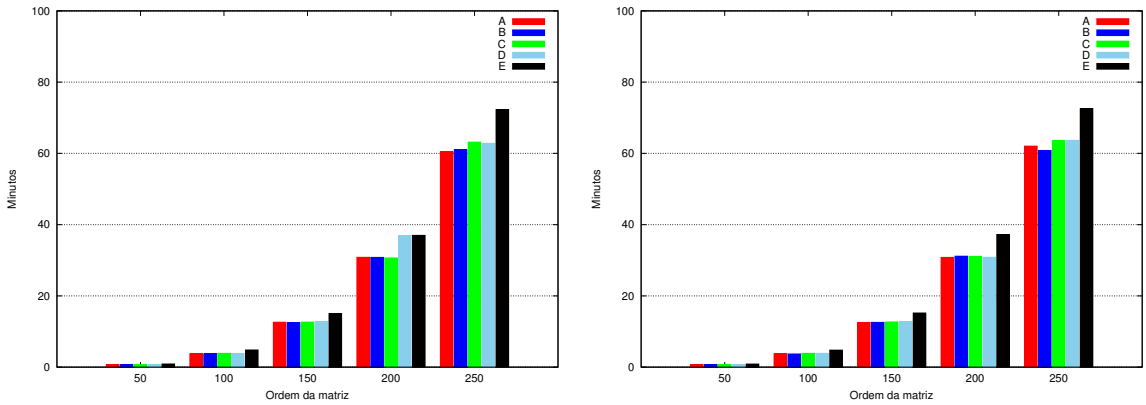

Figura 6 – Execução das matrizes Densas no *Apache Hadoop* com três e quatro nós, respectivamente.

Em uma analise gráfica das Figuras 5 e 6, que mostram os tempos de execução das matrizes, é possível ver que as mudanças em questão de tempo são muito pequenas.

Os resultados para os experimentos A, B, C e D têm claramente, apenas olhando para as Figuras, uma mudança inexpressiva de valores. Apenas o experimento E apresenta uma mudança considerável de valores nos gráficos. Para a matriz de ordem 50 os quatro primeiros experimentos

levaram 0.7 minutos enquanto o experimento E precisou de 0.8 minutos. Na matriz de ordem 100 os experimentos A, B, C e D tiveram pequenas diferenças, A teve o tempo de 3.6 minutos, B 3.7 minutos, C e D 3.8 minutos; por fim o experimento E levou 4.7 minutos. Para a matriz de ordem 150 A e B levaram 12.2 minutos, C 12.3 minutos, D 12.4 minutos enquanto E levou 14.8 minutos. Para a matriz de ordem 200 A, B e D levaram 30.1 minutos e C 29.8 minutos, já E levou 36.5 minutos. Na ultima matriz de ordem 250 A levou 59, 6 minutos, B 59.8 minutos, C 61.9 minutos ,D 61,8 minutos e E 71.1 minutos.

Tabela 2 – Resultados da execução em minutos para o experimento A no *Apache Hadoop*

Ordem da Matriz	**1 nó**	**2 nó**	**3 nós**	**4 nós**
50	0,74	0.74	0.74	0.74
100	3,70	3.71	3.72	3.72
150	12.28	12.41	12.60	12.54
200	30.27	30.52	30.83	30.80
250	59.39	60.26	61.00	62.02
Tempco de execução em minutos				

Tabela 3 – Resultados da execução em minutos para o experimento B no *Apache Hadoop*

Ordem da Matriz	**1 nó**	**2 nó**	**3 nós**	**4 nós**
50	0.74	0.74	0.74	0.74
100	3.68	3.69	3.71	3.68
150	12.28	12.37	12.46	12.55
200	30.19	30.53	30.80	31.1
250	59.64	60.1	61.00	60.81
Tempo de execução em minutos				

Tabela 4 – Resultados da execução em minutos para o experimento C no *Apache Hadoop*

Ordem da Matriz	**1 nó**	**2 nó**	**3 nós**	**4 nós**
50	0.74	0.74	0.74	0.74
100	3.81	3.82	3.85	3.85
150	12.42	12.53	12.63	12.67
200	29.88	30.18	30.65	31.08
250	61.95	62.12	63.15	63.54
Tempo de execução em minutos				

Esses resultados são referentes a execução do *Apache Hadoop* com apenas um nó. As porcentagem de diferença entre os resultados dos experimentos A, B, C e D são muito pequenas, não chegando na maioria das vezes a ser um porcento. No entanto, o experimento E chega ser de 19,25% na matriz de ordem 250. Levando em consideração que o experimento E tem o arquivo

Tabela 5 – Resultados da execução em minutos para o experimento D no *Apache Hadoop*

Ordem da Matriz	**1 nó**	**2 nó**	**3 nós**	**4 nós**
50	0.74	0.74	0.74	0.74
100	3.85	3.84	3.85	3.86
150	12,39	12.59	12.71	12.68
200	30.01	30.4	36.86	30.83
250	61.84	71.96	62.82	63.64
Tempo de execução em minutos				

Tabela 6 – Resultados da execução em minutos para o experimento E no *Apache Hadoop*

Ordem da Matriz	**1 nó**	**2 nó**	**3 nós**	**4 nós**
50	0.85	0.85	0.86	0.86
100	4.78	4.78	4.81	4.76
150	14.83	19.94	15.06	15.17
200	36.59	36.74	36.98	37.22
250	71.14	70.00	72.27	72.51
Tempo de execução em minutos				

de maior tamanho como entrada, pode-se, confirmar que o tempo de leitura exerce influência no desempenho. Os resultados para todos os nós estão sumarizados nas Tabelas 2,3, 4, 5 e 6.

Quando os valores são vistos em número nas Tabelas 2,3, 4, 5 e 6. Mostram que as mudanças são pequenas porém não desprezíveis.

Outro ponto a ser observado quando trata-se de análise de desempenho de um *cluster* é o quanto o desempenho se altera ao adicionar-se mais nós ao *cluster*.No entanto, ao analisar os gráficos representados nas Figuras 5 e 6 e nas Tabelas 2,3, 4, 5 e 6. é possível perceber não há alterações significativas de desempenho em nenhum cenário de experimentos e tamanhos de matrizes.

A configuração usada no *cluster* nesta fase dos experimentos para o bloco do sistema de arquivos HDFS foi a padrão de 128MB. Ao contrário do *OpenMPI* onde o número de tarefas executada pelos nós do *cluster* é passado como parâmetros, no *Apache Hadoop* o número de tarefas iniciadas é o resultado da divisão do tamanho do arquivo de entrada pelo o tamanho do bloco. Os arquivos usados no experimentos A, B, C e D variam de 28KB à 1.6MB, obviamente muito menores que 128MB. Dessa maneira para todas as execuções desses experimentos o *Apache Hadoop* iniciou apenas uma tarefa. Já no experimento E o tamanho dos arquivos variam de de 3.6MB à 517.7MB, sendo que apenas os arquivos referentes as matrizes de ordem 200 e 250 ultrapassam os 128MB. Assim, foram lançadas mais de uma tarefa apenas para essas matrizes.

6.2.2 Resultados da segunda fase de testes

Os experimentos realizados nesta seção levam em consideração a execução do algoritmo de Aytekin (2015) para a multiplicação de matrizes esparsas (ver Seção 4.1.3). Acredita-se que esta implementação seja eficaz na multiplicação de matrizes pois é possível armazenar apenas as linhas e colunas que contenham números diferentes de zero usando a representação dos elementos das matrizes descrita na Seção 4.3.

O ganho de desempenho proporcionado pelo algoritmo de Aytekin (2015) deve-se ao fato de os valores nulos das matrizes esparsas não serem armazenados e consequentemente não são utilizados nos cálculos. Isso faz com que uma quantidade enorme de multiplicações que resultam em zero sejam realizadas.

Os experimentos escolhidos nesta fase foram A, D e E. B e C foram excluídos pois na forma que os arquivos estão salvos os valores nulos precisariam serem utilizados nos cálculos.

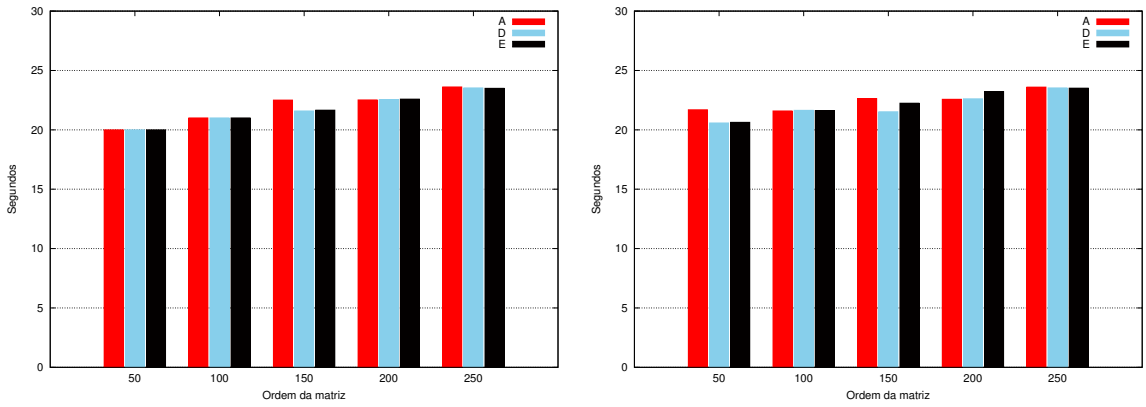

Figura 7 – Execução das matrizes esparsas no *Apache Hadoop* com um nó e dois nós, respectivamente.

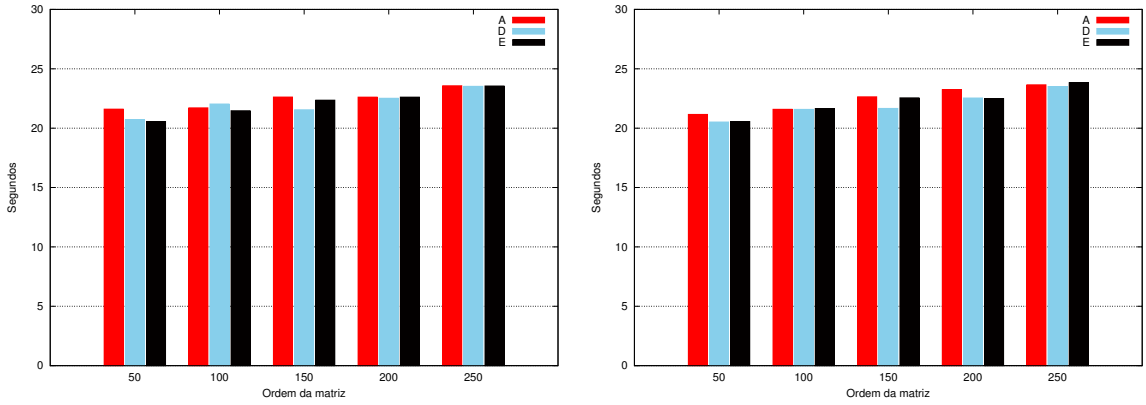

Figura 8 – Execução das matrizes esparsas no *Apache Hadoop* com três e quatro nós, respectivamente.

As ordens das matrizes selecionadas para essa fase de testes foram novamente 50, 100, 150, 200 e 250. Porém, uma medida adicional é necessária quando trata-se de matrizes esparsas,

essa medida é a densidade da matriz. A densidade escolhida foi de 10%, logo as matrizes tiveram 5, 10, 15, 20 e 25 elementos não nulos.

Na execução das matrizes os resultados foram de 20 segundos para matriz de ordem 50, 21 segundos para matrizes de ordem 100, 22 segundos para matrizes de ordens 150 e 200 e 23 segundos para matrizes de ordem 250. Esses valores são referentes aos resultados com apenas um nó.

Tabela 7 – Resultados da multiplicação de matrizes esparsas no *Apache Hadoop*

Ordem da Matriz	**Experimento**	**1 nó**	**2 nós**	**3 nós**	**4 nós**
50	A	20	21.69	21.60	21.16
	D	20,47	20.58	20.71	20.52
	E	20,55	20.63	20.55	20.56
100	A	21.56	21.59	21.71	21.59
	D	21.49	21.64	22.03	21.58
	E	21.46	21.62	21.47	21.64
150	A	22.50	22.63	22.62	22.64
	D	21.58	21.52	21.55	21.66
	E	21.65	22.24	22.46	22.55
200	A	22.51	22.57	22.61	23.24
	D	22.53	22.61	22.52	22.54
	E	22.58	23.21	22.61	22.50
250	A	23.60	23.60	23.57	23.65
	D	23.52	23.52	23.52	23.51
	E	23.49	23.51	23.55	23.83
Tempo de execução em segundos					

Novamente, como pode ser visto nas Figuras 7 e 8 e na sumarização da Tabela 7, a adição de nós não causou uma mudança significativa de desempenho, com diferenças de no máximo 1 segundo.

6.3 Resultados da execução com *OpenMPI*

Uma outra parte dos experimentos deste trabalho foi realizar a execução do algoritmo de Barney (2005) para multiplicação de matrizes usando MPI através do *OpenMPI*. Nessa parte do experimentos, as execuções foram divididas em duas partes, na primeira usando uma implementação em C e outra em Java.

A razão da comparação entre as implementações distintas serve para medir o quanto uma linguagem de programação de mais alto nível são relevantes para o desempenho do algoritmo, isso é linguagens com abstração mais alta do código-fonte em relação ao código de máquina, tais como Java. Além disso, com esse resultado pode-se especular o quanto o fato de ser escrito em Java influência o desempenho do *framework Apache Hadoop*.

Assim como nas execuções no *Apache Hadoop*, cada teste foi executado 12 vezes, e o maior e menor valor foram excluídos do cálculo de média. As ordens de matrizes utilizadas foram 100, 200, 300, 400 e 500. Em ambas as implementações foram executas exatamente as mesmas raízes.

6.3.1 Resultados para a implementação na linguagem C

Na execução com um nó para a matriz de ordem 100 levou 0.32 segundo. A matriz de ordem 200 levou 1 segundo, de ordem 300 levou 2.74 segundos, de ordem 400 levou 6.29 segundos e a matriz de 500 levou 13.37 segundos.

Para as execuções com dois nós os resultados foram: A matriz de ordem 100 levou 0.34 segundo, representando um aumento de 0.02 segundo. A matriz de ordem 200 levou 0.7 segundo, apresentando um ganho de 30%. Na matriz de ordem 300 o tempo foi de 1.8 segundos, com um ganho de 34%. Na matriz de ordem 400 o resultado foi 3.68 segundos, representando um ganho de 40.50%. Para a matriz de ordem 500 foram necessários 6.82, levando a um ganho de 49% em desempenho.

Após a adição do terceiro nó, os resultados foram: A matriz de ordem 100 levou 0.34 segundo, tendo um resultado igual ao segundo nó e 0.02 segundo mais lento que a execução com apenas um nó. Na matriz de ordem 200 o resultado também foi igual a execução com dois nós e com um ganho de 30% em relação a execução de um nó. Para a matriz de ordem 300 foi necessário 1.49 segundos, representando um ganho de 12.22% para a execução com dois nós e 49.27% em relação a execução de um nó. A matriz de ordem 400 teve 2.88 segundos como tempo de execução, com isso houve um ganho de 21.73% em relação a execução com dois nós e 54.21% em relação a execução com um nó. Para execução da matriz de ordem 500 o tempo foi de 5.11 segundos, representando assim, um ganho de 25% em relação a execução com dois nós e 61.37% para a execução com apenas um nó.

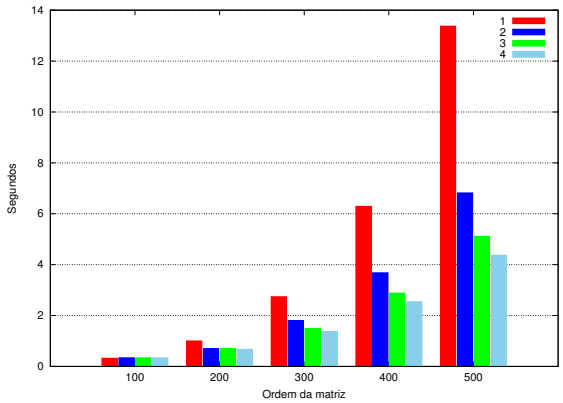

Figura 9 – Execução das matrizes densas usando *OpenMPI* com implementação na linguagem C

Com a inserção do quarto e ultimo nó no *cluster*, os resultados foram: Para a matriz de ordem 50 o tempo gasto foi exatamente igual as execuções com dois e três nós e 0.02 segundo

mais lento que a execução com um nó. Na execução da matriz de ordem 200 o tempo gasto foi 0.67 segundo, apresentando um ganho de apenas 0.03 segundo em relação as execuções com dois e três nós, e um ganho de pouco mais que 30% em relação a execução com um nó. Para a matriz de ordem 300 os resultados começam a apresentar mudanças reais em relação a quantidade de nós, o tempo foi de 1,37 segundo, assim representando um ganho de 7.3% em relação a execução com três nós, de 23.8% em relação a execução com dois nós e 50% em relação a execução com um nó. A matriz de ordem 400 teve 2.54 segundos como seu tempo de execução, isso representa um ganho de 11.8% em relação a execução com três nós, 31% em relação a execução com dois nós e 59.61% em relação a execução com um nó. Para a matriz de ordem 500 o tempo gasto foi de 4.37 segundos, implicando em um ganho de 15.48% em relação a execução com três nos, 36% em relação a dois nós e 67.31% em relação a execução com um nó.

Tabela 8 – Resultados para a multiplicação de matrizes na linguagem C usando *OpenMPI*

Ordem da Matriz	**1 nó**	**2 nós**	**3 nós**	**4 nós**
100	0.32	0.34	0.34	0.34
200	1	0.7	0.7	0.67
300	2.74	1.8	1.49	1.37
400	6.29	3.68	2.88	2.54
500	13.37	6.82	5.11	4.37
Tempo de execução em segundos				

Ao contrário do acontecido as Figuras 5 e 6 que representam graficamente o desempenho do *Apache Hadoop* onde não houve ganho de desempenho após a adição de cada nó, onde não houve ganho real de desempenho. A Figura 9 mostra uma queda muito significativa no ganho de desempenho, chegando a um pico de 67.31% de ganho ao comparar a execução de 1 e 4 nós. Também fica evidente que após uma queda acentuada com a adição do segundo nó as quedas representadas pelas linhas referentes ao terceiro e quarto nó são bem menores e indicam que há um limite de ganhos com adição de novos nós. Os dados da Tabela 8 confirmam o apresentado na Figura 9.

6.3.2 Resultados para a implementação na linguagem Java

Para a execução do algoritmo de multiplicação usando MPI em Java com apenas um nó os respectivos tempos gastos foram: Para a matriz de ordem 100 foram necessários 9.86 segundos, um valor 31.86 vezes maior do que o tempo gasto na execução em C. Para a execução da matriz de ordem 200 o tempo foi de 42.13 segundos, sendo assim 42 vezes mais lento que a execução em C. A matriz de ordem 300 necessitou de 102.76 segundos para ser executada, um valor 37.5 vezes superior à comparação com C. Para a matriz de ordem 400 levou 197.61 segundos, um valor 31.25 vezes maior que o tempo utilizado em C. E por ultimo para a matriz de ordem 500, o tempo gasto foi de 332.02 segundos, um valor 24.83 maior que o tempo em C.

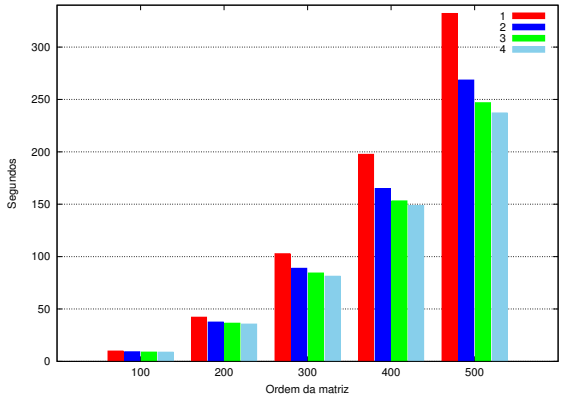

Figura 10 – Execução das matrizes densas usando *OpenMPI* com implementação na linguagem Java

Com a adição do segundo nó, os tempos resultantes foram de: 9.15 segundos para a matriz de ordem 100, um valor 7.2% mais rápido que a execução com dois nós e 26.9 vezes mais lento do que a execução em C para a mesma ordem e mesma quantidade de nós. Para a matriz de ordem 200 o tempo total foi de 37.48 segundos, sendo então 10.76% mais rápido em relação a execução com apenas um nó e 53.54 vezes mais lento do que C. O tempo de multiplicação para a matriz de ordem 300 foi de 88.81 segundos, implicando em um ganho de 13.57% em relação a execução com um nó e 49.31 vezes mais lento do que C. A matriz de ordem 400 levou 165.02 segundos para ser executada, representando um ganho de 14.32% em relação a execução com um nó e 44.84 vezes mais lento que a execução em C. Na ultima matriz, de ordem 500, o tempo for de 268.52 segundos, assim tendo um ganho de 19.12% em relação a execução com um nó e 39.37 vezes mais lento do que C.

Tabela 9 – Resultados para a multiplicação de matrizes na linguagem Java usando *OpenMPI*

Ordem da Matriz	**1 nó**	**2 nós**	**3 nós**	**4 nós**
100	9.86	9.14	8.89	8.82
200	42.13	37.48	36.41	35.57
300	102.76	88.81	84.32	81.26
400	197.61	165.02	153.04	148.64
500	332.02	268.52	246.90	237.07
Tempo de execução em segundos				

As execuções com três nós tiveram os seguintes resultados: A matriz de ordem 100 levou 8.82 segundos, um valor 2.84% menor em relação a execução com dois nós, e 9.8% menor que a execução com um nó, sendo 26.14 vezes mais lenta que a execução em C. Para a matriz de ordem 200 o tempo foi de 36.41 segundos, assim tendo um ganho de 2,5% e 14.14% em relação as execuções com dois nós e um nó respectivamente e 52 vezes mais lento que a execução em C. Para a execução de ordem 300 levou 84.32 segundos, com um ganho de 5% em relação a execução com dois nós e 17.94% em relação a execução com um nó, sendo 56.59 vezes mais

lento que a implementação em C. O tempo necessário para a execução da matriz de ordem 400 foi 153.14 segundos, sendo assim 7.19% mais rápido que a execução para dois nós e 22.50% mais rápido que a execução de um nó, foi 53.17 vezes mais lento do que em C.

Apos a adição do quarto e ultimo nó, os resultados foram: 8.82 segundos para a matriz de ordem 100, significando um ganho de de 0.7% em relação a execução três nós, 3.6% em relação a execução com 2 nós e 10.54% em relação a execução com um nó, além de ter sido 25,9 vezes mais lento que a execução em C. Para a matriz de ordem 200 o tempo resultante foi de 35,57 segundos, tendo um ganho de 2.3% em relação a execução com três nós, 5% em relação a execução com dois nós e 15.57% em relação a execução com um nó, também foi 53 vezes mais lento que a execução em C. Para a execução na matriz de ordem 300 foram necessários 81.26 segundos, assim o ganho foi de 3.6% em relação a execução com 3 nós, 8.5% em relação a execução com dois nós e 20.9% em relação a execução com um nó, também foi 59.31 vezes mais lento do que a implementação em C. Para a matriz de ordem 400 o tempo foi de 148.64 segundos, tendo um ganho de 2.9% em relação a execução com três nós, 10% em relação a dois nós e 22.8% em relação a execução com um nó, e por fim foi 58.47 vezes mais lento que a execução em C. Na matriz de ordem 500 o tempo de execução foi 237 segundos, tendo um ganho de 4% em relação a execução de três nós, 11.7% em relação a execução com dois nós e 28.6% em relação a execução com um nó, por fim foi 54.23 vezes mais lento do que C.

Tabela 10 – Comparativo dos resultados obtidos nas linguagens C e Java usando *OpenMPI*

Ordem da Matriz	**Linguagem**	**1 nó**	**2 nós**	**3 nós**	**4 nós**
100	C	0.32	0.34	0.34	0.35
	Java	9.86	9.14	8.89	8.82
200	C	1	0.7	0.7	0.67
	Java	42.13	37.48	36.41	35.57
300	C	2.74	1.8	1.49	1.37
	Java	102.76	88.81	84.32	81.26
400	C	6.29	3.68	2.88	2.54
	Java	197.61	165.02	153.04	148.64
500	C	13.37	6.82	5.11	4.37
	Java	332.02	268.52	246.90	237.07
Tempo de execução em segundos					

O gráfico da Figura 10 mostra uma queda de tempo com a adição de nó menos acentuada que as quedas identificadas na Figura 9. Enquanto o pico máximo de queda de tempo de um para quatro nós em C foi de 67.31%, na execução em Java o pico foi de 28.6%. Também foi identificado uma queda acentuada em relação de desempenho entre as duas linguagens. Java foi entre 25,9 e 59.31 vezes mais lenta em questão de tempo de execução do que C. A tabela 9 mostra os dados referentes a Figura 10 e a Tabela 10 compara os resultados das linguagem C e Java.

7

Conclusão

Este trabalho investigou a viabilidade de construção de *clusters* de baixo custo para Computação de Alto Desempenho usando computadores *Raspberry PI*. Para realizar a construção dos experimentos foi realizada uma Revisão Sistemática acerca do uso de *Raspberry PI* como ferramenta para Computação de Alto Desempenho. Dentre os trabalhos selecionados na revisão os que trouxeram maior interesse para a investigação realizada foram os trabalhos de Qureshi et al. (2016), Ramos, Ralha e Teodoro (2016) e Lima, Moreno e Dias (2016).

Os trabalhos de Qureshi et al. (2016) e Ramos, Ralha e Teodoro (2016), utilizam um *cluster* formado por *Raspberry PI* para realizar processamento em larga escala de imagens usando, respectivamente, o *framework Apache Hadoop* e a ferramenta *OpenMPI*. Enquanto o trabalho de Lima, Moreno e Dias (2016) realiza cálculos matemáticos usando, também, *OpenMPI*. Dentre os cálculos realizados estão a multiplicação de matrizes.

Na Revisão Sistemática realizada não foi encontrado nenhum trabalho que usasse o *framework Apache Hadoop* para realizar a operação de multiplicação de matrizes. Desta forma, a multiplicação de matrizes foi selecionada como objetivo de investigação neste trabalho. Sendo assim, o *framework Apache Hadoop* e a ferramenta *OpenMPI* foram escolhidos como implementações de dois diferentes *clusters* usando quatro placas *Raspberry PI*.

Desta forma, os experimentos foram pautados na comparação de uma implementação do algoritmo de multiplicação de matrizes usando o modelo de programação do *Apache Hadoop* definido em Apache Software Foundation (2017), enquanto uma outra implementação usada usava o padrão MPI implementado pelo *OpenMPI*. Os algoritmos utilizados foram codificações disponibilizadas por outros autores, sendo eles propostos por Aytekin (2015) e Barney (2005).

Além da métrica mais comum de comparação entre os algoritmos que é o tempo de execução. Este trabalho buscou levar em consideração, o tempo de entrada e de saída, a queda de tempo com a adição de nós. O último tópico de investigação foi como uma linguagem de programação pode influenciar o tempo de execução de um mesmo algoritmo. Para isso o

algoritmo dde Barney (2005) foi implementado nas linguagens Java e C.

Os resultados apresentados por Liang et al. (2014), comparando os resultados do *Apache Hadoop* com uma implementação do MPI para permitir o uso de funções de *Map* e *Reduce* já indicavam uma perda significativa do *Hadoop* em termos de tempo de execução. A hipótese levantada neste trabalho que as diferenças seriam acentuadas pela implementação do *OpenMPI* somado ao algoritmo de multiplicação usando *MapReduce* foi confirmada.

As diferenças chegaram a milhares de segundos entre a implementação usando *Hadoop* e a implementação usando o *OpenMPI* em C. Quando comparado a implementação do *Hadoop* com a linguagem Java, os resultados não foram muito diferentes. Quanto ao ganho de desempenho com a adição de nós não foi constatado ganhos no *Apache Hadoop* , mas o ganho foi significativo quando adicionado no *OpenMPI*, o mesmo acontece para C e Java. Os resultados mostraram que o tempo de entrada e saída no sistema de arquivos HDFS do *Apache Hadoop* não influenciou os resultados significativamente.

Os resultados da comparação de um mesmo algoritmo escrito em linguagens diferentes mostraram uma diferença muito grande tanto em termos de tempo de execução como em ganho de desempenho com a adição de nós. Também mostram que a cada adição de nós o ganho é menor indicando que em alguma quantidade de nós o ganho deve deixar de existir.

Com base nos resultados obtidos teoricamente com a Revisão Sistemática e experimentais obtidos das implementações de algoritmos de multiplicação de matriz. É possível concluir que o *Raspberry PI* pode ser encarado como uma alternativa séria para uma Computação de Alto Desempenho com um baixo custo.

Mesmo com os resultados negativos não se pode descartar o *Framework Apache Hadoop* como uma opção para diferentes algoritmos paralelos. E, principalmente, como um Sistema Distribuído de baixo custo dado sua robustez e tolerância a falha.

Portando, a partir deste trabalho é possível propor como trabalhos futuros, a investigação acerca do ganho de desempenho com novos nós no *Apache Hadoop*, um algoritmo para multiplicação de matrizes esparsas usando *OpenMPI*, a comparação dos resultados obtidos no *Raspberry PI* com um *cluster* convencional e, principalmente, com outras placas de arquitetura ARM, especialmente a placa *Parallella* que conta com um FPGA integrado.

Referências

ANTIQUETECH. *The ENIAC vs. The Cell Phone*. 2013. Disponível em: <http://www.antiquetech.com/?page_id=1438>. Citado na página 15.

Apache Software Foundation. *Apache Hadoop*. 2017. Disponível em: <https://hadoop.apache.org>. Citado 2 vezes nas páginas 30 e 82.

AYTEKIN, M. *Matrix Multiplication with MapReduce*. 2015. Disponível em: <https://lendap.wordpress.com/2015/02/16/matrix-multiplication-with-mapreduce/>. Citado 11 vezes nas páginas 10, 53, 54, 58, 59, 60, 61, 72, 73, 76 e 82.

BAPPALIGE, S. P. *An introduction to Apache Hadoop for big data*. 2014. Disponível em: <https://opensource.com/life/14/8/intro-apache-hadoop-big-data>. Citado 2 vezes nas páginas 30 e 31.

BARNEY, B. *MPI Matrix Multiply - C Version*. 2005. Disponível em: <https://computing.llnl.gov/tutorials/mpi/samples/C/mpi_mm.c>. Citado 7 vezes nas páginas 10, 58, 64, 65, 77, 82 e 83.

CALLIOLI, C.; DOMINGUES, H.; COSTA, R. *Algebra linear e aplicações*. [S.l.]: Atual, 1987. Citado 2 vezes nas páginas 46 e 49.

CORMEN, T. H. et al. *Introduction to Algorithms, Third Edition*. 3rd. ed. [S.l.]: The MIT Press, 2009. ISBN 0262033844, 9780262033848. Citado na página 50.

COULOURIS; DOLLIMORE, J.; KINDBERG, T. *Distributed Systems: Concepts and Design (4th Edition) (International Computer Science)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005. ISBN 0321263545. Citado na página 25.

DEAN, J.; GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, ACM, New York, NY, USA, v. 51, n. 1, p. 107–113, jan. 2008. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1327452.1327492>. Citado 3 vezes nas páginas 30, 51 e 52.

HOFFMAN, A. R. et al. *Supercomputers: directions in technology and applications*. Washington,DC: National Academies, 1990. 122 p. ISBN 0309040884. Citado na página 16.

KECSKEMETI, G.; HAJJI, W.; TSO, F. P. Modelling low power compute clusters for cloud simulation. In: *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. [S.l.: s.n.], 2017. p. 39–45. Citado 2 vezes nas páginas 35 e 43.

KITCHENHAM, B. Procedures for performing systematic reviews. v. 33, 08 2004. Citado 2 vezes nas páginas 18 e 32.

LEON, S. *Álgebra linear com aplicações*. [S.l.]: LTC, 1999. ISBN 9788521611509. Citado 2 vezes nas páginas 46 e 49.

LEVORATO, G. B. P. *Matrizes, Determinantes e Sistemas Lineares: Aplicações na Engenharia e Economia*. Dissertação (Mestrado) — Universidade Estadual Paulista Júlio de Mesquita Filho, 2017. Citado na página 45.

LIANG, F. et al. Performance benefits of datampi: A case study with bigdatabench. In: *Big Data Benchmarks, Performance Optimization, and Emerging Hardware - 4th and 5th Workshops, BPOE 2014, Salt Lake City, USA, March 1, 2014 and Hangzhou, China, September 5, 2014, Revised Selected Papers*. [s.n.], 2014. p. 111–123. Disponível em: <https://doi.org/10.1007/978-3-319-13021-7_9>. Citado 3 vezes nas páginas 56, 57 e 83.

LIMA, F. A.; MORENO, E.; DIAS, W. R. A. Performance analysis of a low cost cluster with parallel applications and arm processors. *IEEE Latin America Transactions*, v. 14, n. 11, p. 4591–4596, Nov 2016. ISSN 1548-0992. Citado 8 vezes nas páginas 24, 25, 31, 40, 41, 43, 44 e 82.

MADURANGA, M. W. P.; RAGEL, R. G. Comparison of load balancing methods for raspberry-pi clustered embedded web servers. In: *2016 International Computer Science and Engineering Conference (ICSEC)*. [S.l.: s.n.], 2016. p. 1–4. Citado 2 vezes nas páginas 36 e 43.

MAPPUJI, A. et al. Study of raspberry pi 2 quad-core cortex-a7 cpu cluster as a mini supercomputer. In: *2016 8th International Conference on Information Technology and Electrical Engineering (ICITEE)*. [S.l.: s.n.], 2016. p. 1–4. Citado 3 vezes nas páginas 37, 43 e 58.

MOORE, J. *Performance Benchmarking a Raspberry PI Cluster*. 2014. Disponível em: <https://opensky.ucar.edu/islandora/object/siparcs3A132/datastream/PDF/download/citation.pdf>. Citado na página 37.

OPENMPI. *Open MPI: Open Source High Performance Computing*. 2018. Disponível em: <https://www.open-mpi.org/>. Citado na página 31.

PACHECO, P. *An Introduction to Parallel Programming*. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 9780123742605. Citado 2 vezes nas páginas 28 e 29.

PAHL, C. et al. A container-based edge cloud paas architecture based on raspberry pi clusters. In: *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*. [S.l.: s.n.], 2016. p. 117–124. Citado 2 vezes nas páginas 35 e 43.

PEDRINI, H.; SCHWARTZ, W. *Análise de imagens digitais: princípios, algoritmos e aplicações*. THOMSON PIONEIRA, 2008. ISBN 9788522105953. Disponível em: <https://books.google.com.br/books?id=13KAPgAACAAJ>. Citado 2 vezes nas páginas 38 e 45.

PFISTER, G. F. *In Search of Clusters (2Nd Ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998. ISBN 0-13-899709-8. Citado na página 16.

PRECHELT, L. An empirical comparison of seven programming languages. *Computer*, v. 33, n. 10, p. 23–29, Oct 2000. ISSN 0018-9162. Citado na página 64.

QURESHI, B. et al. Performance of a low cost hadoop cluster for image analysis in cloud robotics environment. *Procedia Computer Science*, v. 82, n. Supplement C, p. 90 – 98, 2016. ISSN 1877-0509. 4th Symposium on Data Mining Applications, SDMA2016, 30 March 2016, Riyadh, Saudi Arabia. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1877050916300278>. Citado 8 vezes nas páginas 37, 43, 44, 53, 57, 61, 64 e 82.

RAMOS, R.; RALHA, C. G.; TEODORO, G. Avaliação de cluster raspberry pi para execução de aplicações de análise de imagens microscópicas médicas. In: *43º Semish - Seminário Integrado de Software e Hardware*. [S.l.: s.n.], 2016. p. 1795–1806. Citado 7 vezes nas páginas 39, 43, 44, 57, 61, 64 e 82.

RAUBER, T.; RNGER, G. *Parallel Programming: For Multicore and Cluster Systems*. 2nd. ed. [S.l.]: Springer Publishing Company, Incorporated, 2013. ISBN 3642378005, 9783642378003. Citado 3 vezes nas páginas 21, 29 e 31.

SALE, T. *Virtual Wartime Bletchley Park*. 2010. Disponível em: <http://www.codesandciphers.org.uk/virtualbp/tbombe/tbombe.htm>. Citado na página 15.

SEAL, D. *ARM Architecture Reference Manual*. 2nd. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN 0201737191. Citado na página 24.

SIPSER, M. *Introduction to the Theory of Computation*. 1st. ed. [S.l.]: International Thomson Publishing, 1996. ISBN 053494728X. Citado na página 15.

SQUYRES, J. *Java Bindings for Open MPI*. 2014. Disponível em: <https://blogs.cisco.com/performance/java-bindings-for-open-mpi>. Citado na página 67.

STALLINGS, W. *Computer Organization and Architecture*. Pearson-Prentice Hall, 2015. (Always learning). ISBN 9780134101613. Disponível em: <https://books.google.com.br/books?id=BtnKrQEACAAJ>. Citado 3 vezes nas páginas 22, 23 e 24.

TANENBAUM, A. S. *Structured Computer Organization (5th Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2005. ISBN 0131485210. Citado 5 vezes nas páginas 21, 22, 23, 24 e 27.

TANENBAUM, A. S.; BOS, H. *Modern Operating Systems*. 4th. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014. ISBN 013359162X, 9780133591620. Citado 6 vezes nas páginas 16, 19, 20, 21, 28 e 50.

TANENBAUM, A. S.; STEEN, M. v. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006. ISBN 0132392275. Citado 2 vezes nas páginas 25 e 26.

TSO, F. P. et al. The glasgow raspberry pi cloud: A scale model for cloud computing infrastructures. In: *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops*. [S.l.: s.n.], 2013. p. 108–112. ISSN 1545-0678. Citado 3 vezes nas páginas 34, 35 e 43.

WIDRIKSSON, J. *Raspberry PI Hadoop Cluster*. 2014. Disponível em: <http://www.widriksson.com/raspberry-pi-hadoop-cluster>. Citado na página 53.

Apêndices

APÊNDICE A – Instalação do Cluster Hadoop Raspberry PI

Para a realização dos experimentos do Trabalho de Conclusão de Curso foi utilizado um *cluster* baseado no *framework Apache Hadoop*, cujo os parâmetros de instalação serão mostrados neste apêndice.

A.1 Sistema Operacional

Todos os *Raspberry* do *cluster* rodaram o *Raspbia-Stretch Lite* como Sistema Operacional. A escolha do *Raspbian* se deu pelo fato de contar com o maior repositório disponível de softwares *ARM*. E a versão *Lite* por ser uma versão mais enxuta do sistema.

A.2 *Softwares* Usados

- OpenSSH - já instalado por padrão no *Raspbian*
- Open JDK 8
- Apache Hadoop 3.0.0-Alpha4

A.3 Instalação

1. Instalando o JDK java

```
1 sudo apt-get install java-oracle-jdk8
```

2. Preparar o usuário e o grupo de usuário do *Hadoop*.

```
1 sudo addgroup hadoop
2 sudo adduser --ingroup hadoop hduser
3 sudo adduser hduser sudo
```

3. Baixar e instalar o apache Hadoop3.0.0-Alpha4

```
wget http://apache.mirrors.spacedump.net/hadoop/core/hadoop-3.0.0-alpha4/
hadoop-3.0.0-alpha4.tar.gz
sudo mkdir /opt
sudo tar -xvzf hadoop-3.0.0-alpha4.tar.gz -C /opt/
cd /opt
sudo mv hadoop-3.0.0-alpha4 hadoop
sudo chown -R hduser:hadoop hadoop
```

4. Editar arquivos /etc/host para que todos os *Raspberry* se comuniquem.

Código 8 – /etc/hosts

```
master 192.168.0.1
slave1 192.168.0.2
slave2 192.168.0.3
slave3 192.168.0.4
slave4 192.168.0.5
```

5. Gerar as chaves ssh em todos os *Raspberry*.

6. Adicionar as variáveis de ambiente do *Hadoop* ao bashrc

Código 9 – /.bashrc

```
export JAVA_HOME=export
JAVA_HOME=\$(readlink -f /usr/bin/java | sed "s:bin/java::")
export HADOOP_INSTALL=/opt/hadoop
export PATH=$PATH:$HADOOP_INSTALL/bin
```

7. Adicionar o $JAVA_HOME ao fim do /opt/hadoop/etc/hadoop/hadoop-env.sh

Código 10 – hadoop-env.sh

```
export JAVA_HOME=\$(readlink -f /usr/bin/java | sed "s:bin/java::")
```

8. Os passos seguintes mostra os arquivo de configuração do *Hadoop* como foram utilizados nos experimentos.

9. /opt/hadoop/etc/hadoop/core-site.xml

Código 11 – core.site.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
   <property>
       <name>hadoop.tmp.dir</name>
       <value>/hdfs/tmp</value>
   </property>
   <property>
       <name>fs.default.name</name>
       <value>hdfs://master:54310</value>
   </property>
</configuration>
```

10. /opt/hadoop/etc/hadoop/mapred-site.xml

Código 12 – mapred-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
       <name>mapred.job.tracker</name>
       <value>master:54311</value>
  </property>
</configuration>
```

11. /opt/hadoop/etc/hadoop/hdfs-site.xml

Código 13 – hdfs-site.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

12. Criando o sistema de arquivos HDFS.

Código 14 – Comandos para criar o sistema de arquivos

```
sudo mkdir -p /hdfs/tmp
sudo chown hduser:hadoop /hdfs/tmp
sudo chmod 750 /hdfs/tmp
hadoop namenode -format
```

13. Iniciando o *Apache Hadoop*.

Código 15 – Comando para iniciar o *cluster*

```
/opt/hadoop/sbin/start-dfs.sh
/opt/hadoop/sbin/start-yarn.sh
```

14. Verificando a instalção

Código 16 – Comando para verificar a integridade da instalação

```
/opt/hadoop/hadoop/bin/hdfs dfsadmin -report
```

APÊNDICE B – Instalação do *Cluster Raspberry PI* usando MPI

Para a realização dos experimentos do Trabalho de Conclusão de Curso foi utilizado um *cluster* baseado no *OpenMPI*, cujo os parâmetros de instalação serão mostrados neste apêndice.

B.1 Sistema Operacional

Todos os *Raspberry* do *cluster* rodaram o *Raspbian-strech Lite* como Sistema Operacional. A escolha do *Raspbian* se deu pelo fato de contar com o maior repositório disponível de softwares *ARM*. E a versão *Lite* por ser uma versão mais enxuta do sistema.

B.2 *Softwares* Usados

- OpenSSH - já instalado por padrão no *Raspbian*
- Open JDK 8
- Gfortran
- OpenMPI 3.1.0

B.3 Instalação

1. Instalando o JDK java e Gfortran

```
1 sudo apt-get install java-oracle-jdk8 gfortran
```

2. Instalação do OpenMPI

```
https://download.open-mpi.org/release/open-mpi/v3.1/openmpi-3.1.0.tar.gz
wget https://download.open-mpi.org/release/open-mpi/v3.1/openmpi-3.1.0.tar.gz
tar -xf openmpi-3.1.0.tar.gz
cd openmpi-3.1.0
sudo CFLAGS=-march=armv7-a CCASFLAGS=-march=armv7-a  ./configure --prefix=/usr/local
 ↪  --enable-mpi-java --with-jdk-bindir=/usr/lib/jvm/java-8-openjdk-armhf/bin
 ↪  --with-jdk-headers=//usr/lib/jvm/java-8-openjdk-armhf/include
sudo make all install
```

B.4 O *hostfile*

O *Openmpi* precisa de um arquivo que lhe diga quais os ips estão disponíveis e quantas tarefas podem ser executada por cada máquina.

O arquivo a seguir representa este arquivo nessa instalação:

```
192.168.0.2 slots=2
192.168.0.3 slots=1
192.168.0.4 slots=1
192.168.0.5 slots=1
```

Esta configuração permite o mestre executar uma tarefa do escravo.